C Information Bulletin # 1

X3 Project No. 878-D

NOTICE

Recognizing the need for a uniform approach to the responsibility for disseminating interpretations to approved American National Standards, the Accredited National Standards Committee X3 — Information Processing Systems has authorized the publication of Technical Information Bulletins.

This Technical Information Bulletin is issued in response to questions which have been raised regarding certain specifications contained in the content of:

X3.159-1989
American National Standard for Information Systems —
Programming Language C

This Bulletin was prepared by Technical Committee X3J11, which developed that Standard, and was authorized for release by Committee X3 in order to provide interpretations as quickly as possible in response to questions raised.

This Bulletin, while reflecting the technical opinion of the Technical Committee which developed the standard, is intended solely as supplementary information to other users of the standard. That standard, ANS X3.159-1989, as approved through the publication and voting procedures of the American National Standard Institute, is not altered by this Bulletin. Any subsequent revision to the standard, ANS X3.159-1989, may or may not reflect the contents of this Technical Information Bulletin.

May 27, 1992

Editor's Remarks

Apart from the introductory material, this document merely reprints previous requests for interpretation along with Technical Committee X3J11's responses to them. The committee continues to respond to requests for interpretation; it may or may not eventually produce additional Technical Information Bulletins.

For the convenience of users of the International Standard, ISO/IEC 9899:1990, the following table provides a mapping from page and section numbers cited in this document, which refer to ANS X3.159-1989, to their equivalents in ISO/IEC 9899:1990. Within each major section, subsection numbering follows the same scheme for both documents.

ANS X3.159-1989			**ISO/IEC 9899:1990 (E)**		
page	*§*	*title*	*page*	*§*	*title*
1	1	Introduction			
1	1.1	Purpose	1	1	Scope
1	1.2	Scope	1	1	Scope
2	1.3	References	1	2	Normative references
2	1.3	References	177	A	Bibliography
2	1.4	Organization of the Document	vii		Introduction
2	1.5	Base Documents	vii		Introduction
2	1.6	Definition of Terms	2	3	Definitions and conventions
4	1.7	Compliance	3	4	Compliance
5	1.8	Future Directions	vii		Introduction
6	2	Environment	5	5	Environment
19	3	Language	18	6	Language
97	4	Library	96	7	Library
178	A	Language Syntax Summary	178	B	Language syntax summary
189	B	Sequence Points	189	C	Sequence points
190	C	Library Summary	190	D	Library summary
196	D	Implementation Limits	196	E	Implementation limits
198	E	Common Warnings	198	F	Common warnings
199	F	Portability Issues	199	G	Portability issues
210		Index	210		Index

Request for Interpretation # 1 (X3J11 Doc. No. 90-009)

Question 1: Do functions return values by copying?

The standard is clear (in § 3.3.2.2) that function arguments are copied, but is not clear (in § 3.6.6.4) whether a function's returned value is also copied. This question becomes an issue in the assignment statement `s=f();` where `f` yields a structure: is the result defined when the structure `s` overlaps the structure that `f` obtained the returned value from?

I ask this question because the GNU C compiler does not copy the structure in this case. When I filed the enclosed bug report,* Richard Stallman, the author of GNU C, replied that he didn't think that Standard C required the extra copy. I sympathize with Stallman's desire for efficient code, and I also would prefer that the standard did not require the extra copy here, but the point should be made clear in the standard.

Interpretation:

§ 3.6.6.4 (''The `return` Statement'') says ''the value of the expression is returned to the caller as the value of the function call expression.'' If any storage used to hold ''the value'' overlaps storage used for any other purpose, then ''the value'' would not be well-defined. Therefore, no overlap is allowed.

The overlap permission granted in § 3.3.16.1 does not apply to the case of function return. This is based upon understanding of the intent.

* Omitted from this document.

Request for Interpretation # 2 (X3J11 Doc. No. 90-010)

Question 1: § 3.8.3.2: Semantics of #

A minor detail in the semantics is that it does not explicitly state that a \ character will be inserted before a \ character that occurs within a macro actual parameter, only when the \ character occurs within a string literal or character constant within the actual parameter.

I can see that there is an argument concerning the systems where \ is a valid part of a path name and where `#include` path names are desired to be built dynamically and then #ed.

Would it not be better, however, to escape all \ within actual parameters and require either

a) that systems using \ in path names escape them within `#include` strings, or perhaps better require

b) that during macro processing of `#include` parameters the # operator will not cause \ characters to be escaped at all or will be implementation defined?

This second case b) is better, as strings for `#include` directives are already a special case (no escape processing *etc.*), so should not the # operator also be special for `#include` directives?

Interpretation:

This is not a request for interpretation; this is a request for changes to the standard. Changes cannot be considered until the next review of the language. The committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous response to this item from David F. Prosser.* The committee endorses the substance of this response, which follows:

The rules in § 3.8.3.2 regarding \ were discussed in the Committee and the result is as intended. The Committee's charter was to standardize prior art where such was clear, and the behavior of those C preprocessing phases that allowed tokens such as \ left them alone, even when inserting them into strings. However, the Committee also had license to fix or add to the language where it was judged to be deficient. Since none of the existing stringizing preprocessing phases correctly handled string literals and certain character constants, the special rules for these were chosen.

§ 3.8.3's examples (page 93, lines 3-31) include a \ that is outside of a string literal or character constant. If the rules were to be modified along the lines of your proposal, the intended effect would not happen.

One of the main points in your argument is that uncontained \s are only critical in path names that use \ as a special character, and that this is only needed when `#include` filenames are constructed via macro replacement. I agree that the current rules do allow this sort of use without too much trouble, but I don't see this as being a main motivation for this feature. By default, the rules for stringizing were that the original spelling of the invocation argument is placed into a string literal. The only exceptions made to this were those that were valid C tokens that could not be simply inserted between a pair of "s. The rules for \ and " within string literals and character constants were derived from that need only. Furthermore, a lone \ is a preprocessing token due to the ''some other character'' rule of the syntax from § 3.1. This would be the only place where special constraints were placed on one of this type of preprocessing token.

Finally, solution b) of your discussion involves context-dependent rules for the stringizing operation. While there is a minor context dependency regarding macro replacement and the `defined` unary operator on `#if` and `#elif`

* Editor of the standard.

directive lines, this is the only context dependency in the whole set of macro replacement rules. Moreover, this dependency is at the topmost level only. Solution b) would require a flag noting whether the result of the replacement was to be used within a `#include` directive. Therefore, the same macro invocation would produce different results at different invocations. (At the least, debugging and/or testing of a tricky macroized `#include` directive would be more difficult.)

In conclusion, to the best of my knowledge, the Committee wants to keep the behavior here as currently described, and made this choice intentionally.

Request for Interpretation # 3 (X3J11 Doc. No. 90-011)

Question 1: § 3.1.8: Preprocessing numbers

I note from the rationale document of November 1988, X3J11 Document Number 88-151, that the following problem has been observed. I am surprised at the committee's decision to allow such a loose description.

Under the grammar given for a *pp-number*

```
0xEE+23          0x7E+macro       0x100E+value-macro
```

are preprocessing numbers and as such a conforming C compiler would be required to generate an error when it failed to successfully convert them to actual C language number tokens.

The solution is simply to restrict the inclusion of `[eE][+-]` within a *pp-number* to situations where the `e` or `E` is the first *non-digit* in the character sequence composing the preprocessing number. This can be easily implemented in a variety of methods; the informal description above gives perhaps a better guide to efficient implementation than the following revised grammar:

pp-number:
- *pp-float*
- *pp-number digit*
- *pp-number* **.**
- *pp-number non-digit* // A *non-digit* is a letter or underscore

pp-float:
- *pp-real*
- *pp-real* `E` *sign*
- *pp-real* `e` *sign*

pp-real:
- *digit*
- **.**
- *pp-real digit*
- *pp-real* **.**

It is unbelievable that a standards committee could so lose sight of its objective that it would, in full awareness, make simple expressions illegal.

To illustrate the absurdity of the rationale document's claim that the faulty grammar was felt to be easier to implement, why not adopt the following grammar for a *pp-number* and really make life simple; after all, who wants to have their preprocessor slowed down by checking whether the `+` or `-` was preceded by an `e` or an `E`??

pp-number:
- *digit*
- **.**
- *pp-number digit*
- *pp-number* **.**
- *pp-number non-digit*
- *pp-number sign*

Interpretation:

This is not a request for interpretation; this is a request for changes to the standard. Changes cannot be considered until the next review of the language. The committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous responses to this item from David F. Prosser. The committee endorses the substance of these responses, which follow:

In response to your first suggested grammar: This grammar doesn't include all valid numeric constants and exclude other important tokens. For example, `.` is derivable. But let's assume that you intended something like

pp-number:
    pp-float
    digit
    pp-number digit
    pp-number non-digit

pp-float:
    pp-real
    pp-float `E` *sign*
    pp-float `e` *sign*
    pp-float digit
    pp-float `.`
    pp-float non-digit

pp-real:
    digit
    `.` *digit*
    pp-real digit
    pp-real `.`

This grammar is certainly more complicated than the one-level construction in the proposed standard, and consequently harder to understand. That's a strike against it.

Another strike is that, while it does mimic the two major numeric categories, it still doesn't include all sequences covered by the existing grammar, save those that would otherwise be valid by the stricter tokenization rules. For example, `0b0101e+17` (might be someone's future notion of a binary floating constant).

Finally, it suffers from a great deal of reduce/reduce conflicts, making the implementation and specification less likely to be understood and implemented as intended.

In response to your second suggested grammar: This could have been done. But the Committee chose a compromise at a different point — one that restricts the inappropriate gobbling of characters to `+` and `-` immediately after `E` or `e`. This was all that was necessary to cover all valid numeric constants in as simple a grammar as was possible.

For more background, you'd need to know the state of the proposed standard a few years before this grammar was voted in. The Committee had stated its intent that ''garbage'' character sequences that began like a numeric constant were to be tokenized as a single sequence. This was to prevent situations in which this ''garbage'' would be turned into valid C code through obscure macro replacements, among more minor reasons. This was, unfortunately, very poorly stated in the draft. As I recall, it was placed in the constraints for § 3.1. It was

something like ‘‘Each pair of adjacent tokens that are both keywords, identifiers, and/or constants must be separated by white space.’’*

As you can see, this constraint neither presented the intent of the Committee nor caused implementations to behave in any sort of consistent manner with respect to tokenization.

Finally a letter writer understood the issue well enough to suggest a grammar along the lines of the current § 3.1.8. It, contrary to your opening remarks on this topic, is *not* a ‘‘loose description’’, and it finally stated in a precise way the intent of the tokenization rules.

The benefits of this construction were that all tokenization for all implementations would now be the same, no ‘‘garbage’’ character sequences would be able to be converted to valid C code, skipped blocks of code could silently be scanned without generating needless and unnecessary tokenization errors, the preprocessing tokenization of numeric tokens would be greatly simplified, and room for future expansion of C’s numeric tokens was reserved.

That’s a lot of good. The down side was that certain sequences now would require some white space to cause them to be tokenized as the programmer intended. As noted in the rationale document, there are other parts in C that require white space for tokenization to be controlled, and this was found to be one more.

Since the ‘‘mistokenization’’ of such sequences must result in some diagnostic noise from the compiler, and since the fix is so mild, the Committee agreed that the proposed standard is still much better with this grammar than with any of the other suggestions.

Personally, I think that the biggest surprise ‘‘win’’ was the reservation of future numeric token ‘‘name space’’. I would not be at all surprised to find binary constants (that begin with `0b`) in newer C implementations.

Question 2: § 3.8.3: Macro substitutions, tokenization, and white space

In general I think it is a good guiding principle that a C implementation should be able to be based around completely disjoint preprocessing and lexical scanning parses of the compiler. As such the rules on tokenizing need to be emphasized with the following paragraphs (possibly placed after paragraph 1 of § 3.8.3.1):

> All macro substitutions and expanded macro argument substitutions will result in an additional space token being inserted before and after the replacement token sequence where such a space token is not already present and there is a corresponding preceding or subsequent token in the target token sequence.
>
> The last token of every macro argument has no subsequent token at the time of its initial macro argument expansion, and similarly a macro parameter that is the last token of a replacement token list has no subsequent token at the time of that parameter’s substitution. Similarly for first tokens and preceding tokens.

Naturally such a step can be treated as purely conceptual by a tokenized implementation with combined preprocessing and lexical analysis, except for the purposes of argument stringizing where the added spacing may be essential for unambiguous identification of the preprocessing tokens involved.

Such a statement is not a substantive change, as it is merely clarifying the tokenization rules, and given that Standard C has changed the definition of the preprocessor substantially from K&R already (*re* macro argument expansion before substitution) such an additional explicit change from K&R C should cause comparatively little difficulty except to those who had not appreciated just how different the preprocessing rules are already.

---

* As ‘‘improved’’ for the May 1, 1986 draft proposed standard, § 3.1 Constraints consisted of the single sentence: ‘‘Each keyword, identifier, or constant shall be separated by some white space from any otherwise adjacent keyword, identifier, or constant.’’

Examples which are clarified by this change are:

```
#define       macro       +
                    +macro
                    macro+
#define       mac()       +
#define       ro             +
                    mac()ro
```

all of which unambiguously result in lines with two + operator tokens, in strict accordance with the draft standard's tokenization rules, and not, as was formerly the case with traditional text oriented preprocessors, in single ++ operators.

Examples which are changed by this statement are:

```
#define       mac()                +
#define       ro                      +
#define       str(s)               #  s
#define       eval(m,e)         m(e)
                    eval(  str,  mac()ro  )
```

which produces the string "+ +" and not the string "++" as it would do with the draft's current wording.

Interpretation:

This is not a request for interpretation; this is a request for changes to the standard. Changes cannot be considered until the next review of the language. The committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous responses to this item from David F. Prosser. The committee endorses the substance of these responses, which follow:

K&R never specified the macro replacement algorithm to the extent that any such conclusion is possible. The widest range of implementation choices were present in this area of the language. The eventual choice of a macro replacement algorithm was one that did not match any existing implementation, but one that tried to include the behavior of all major variants.

You agree that the proposed standard is clear that once a token is recognized, it is never retokenized unless subjected to a # or ## operation. The behavior described is that which was chosen by the Committee. Your proposal would cause, as you note, certain created string literals to include white space not present in the original text. This runs counter to the # operator's goal of producing a string version of the spelling of the invocation arguments.

The proposed standard allows an implementation that uses a text-to-text separate preprocessing stage the option to use white space as necessary to separate tokens when it produces its output. However, this insertion of white space must not be visible to the program. The proposed extra white space would probably be a surprise to the programmer as well.

Finally, this proposal would require those implementations that have a built-in preprocessing stage to add extra code to insert white space in special circumstances. This is counter to the goal of having both built-in and separate implementations be purely an implementation choice.

Question 3: § 3.8.3: Empty arguments to function-like macros

I would like to make a request for clarification and a request for a stronger statement of standardization.

The request for clarification is: given

```
#define  macro(  xx  )  xx
              macro()
```

is this a constraint violation of § 3.8.3 Constraints paragraph 4:

> The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition, …

or is this an undefined, implementation-dependent program — § 3.8.3 Semantics paragraph 5:

> If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined.

In connection with the above I would request that the committee make a much stronger statement as to whether empty arguments are to be treated as valid arguments or are to be treated as errors. They can have their uses, but if that is recognized then it should be standardized; if not, it should not be allowed.

Interpretation:

If one takes the general case, empty arguments in invocations of function-like macros are easy to recognize:

```
#define  f(a,b,c)  whatever
f(,,)
```

These empty arguments all have ‘‘shadows’’ that cause the sentence you reference in § 3.8.3 (page 91, lines 4-5) to be clearly in effect.

The only uncertain case is one in which an empty argument in an invocation of a one-parameter function-like macro mimics a ‘‘no arguments’’ invocation. (It should also be noted that the definition of a macro argument from § 1.6 does not preclude an empty sequence.)

Thus the standard is clear that the behavior is undefined in the example from your request. If an implementation does not choose to allow empty arguments, a diagnostic will probably be emitted; otherwise, the invocation will most likely be replaced by a preprocessing token sequence in which the parameter is replaced with no tokens. But because the standard does not define this, other than as a common extension, there are no guarantees.

Question 4: § 3.8.3: Preprocessor directives within actual macro arguments

It is a guiding principle that a macro function and an actual function should be invokable in as similar fashion as possible. In the latter case, it is not uncommon to find code with variations of arguments subject to conditional compilation. This should also compile correctly if an appropriate macro definition is made for the function.

While conditional compilations within function arguments is not necessarily a programming style that I would condone, I feel that it is in the interests of the C programming community at large for such constructs to be well formed, even if forbidden, and as such make the following requests:

I would like the committee to change § 3.8.3 to state that `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` preprocessing directives are allowed within actual macro arguments (not necessarily cleanly nested).

Conversely, I would like `#define` and `#undef` to be formally forbidden within macro invocations, as these can result in effects that are dependent on the particular implementation of the macro expansions.

Interpretation:

This is not a request for interpretation; this is a request for changes to the standard. Changes cannot be considered until the next review of the language. The committee reasserts that the grammar and/or semantics of preprocessing as they appear in the standard are as intended.

We are attaching a copy of the previous response to this item from David F. Prosser. The committee endorses the substance of this response, which follows:

The equivalent of your proposal was rejected a couple of years ago. Certain Committee members felt that requiring all preprocessors to recognize these lines as directives was too much. Those that felt that these lines must be recognized were finally convinced that it was enough to allow implementations the right to behave in the more orthogonal manner. (Maybe they figure that the next version of the standard will have to require this sort of behavior, as all ‘‘reasonable’’ implementations already will have it by then.)

Request for Interpretation # 4 (X3J11 Doc. No. 90-012)

Question 1: Are multiple definitions of unused identifiers with external linkage permitted?

The wording in § 3.7 permits multiple definitions of identifiers with external linkage, so long as the identifiers are never used. For example, the following program is ‘‘strictly conforming’’ if you read the wording in § 3.7 literally:

```
int  F()  {return  0;}
int  F()  {return  1;}
int  V  =  0;
int  V  =  1;
int  main()  {return  0;}
```

This must be a bug in the wording of § 3.7. It cannot have been the Committee’s intent, since it prohibits the most commonly encountered linker model. For example, most linkers will flatly refuse to link the following ‘‘strictly conforming’’ program

x.c:

```
int  F()  {return  0;}
int  G(int  i)  {return  i;}
```

y.c:

```
int  F()  {return  1;}
int  G(int);
int  main()  {return  G(0);}
```

because `F` is defined twice.

Interpretation:

This was disambiguated by an editorial change in the final standard. See the attached report of the Redactor, X3J11 Document Number 90-001, page 3, regarding § 3.7:

> **§ 3.7, page 82, line 25 —** BSI #28
>
> Change ‘‘identifier’’ to ‘‘identifier; otherwise there shall be no more than one’’. (This clarifies a possible unintended reading that allows any number of external definitions if the identifier is never used!)

Request for Interpretation # 5 (X3J11 Doc. No. 90-020)

Question 1:

According to § 3.8.6, a pragma directive ‘‘causes the implementation to behave in an implementation-defined manner.’’ May a conforming implementation define and recognize a pragma which would change the semantics of the language? For example, might a conforming implementation recognize and honor the directive

```
#pragma  UNSIGNED_PRESERVING
```

as a way for a program to request non-standard integral promotions?

I also pose the corollary question. May a strictly conforming program contain a pragma directive? According to § 1.7, a strictly conforming program ‘‘shall use only those features of the language … specified in this standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, …’’

If there is no constraint on how a conforming implementation may behave when encountering a pragma directive, would it not follow that a strictly conforming program may not contain a pragma directive?

Interpretation:

The relevant citations are § 3.8.6

> A … `pragma` … causes the implementation to behave in an implementation-defined manner.

and § 1.7

> A strictly conforming program … shall not produce output dependent on any … implementation-defined behavior …

In response to each question:

1) Yes, a conforming implementation may define and recognize a pragma which would change the semantics of the language.
2) Yes, for example, it might honor `UNSIGNED_PRESERVING`.
3) No, a strictly conforming program may not contain a pragma directive.
4) We agree with your conclusion, reasserting answer number 3.

Request for Interpretation # 6 (X3J11 Doc. No. 90-020)

Question 1:

It is unclear how the `strtoul` function behaves when presented with a subject sequence that begins with a minus sign. The `strtoul` function is described in § 4.10.1.6, which contains the following statements.

> If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.
>
> If the correct value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the macro `ERANGE` is stored in `errno`.

Assume a typical 32-bit, two's-complement machine with the following limits.

```
LONG_MIN        -2147483648
LONG_MAX         2147483647
ULONG_MAX        4294967295
```

Assuming that the value of `base` is zero, how should `strtoul` behave (return value and possible setting of `errno`) when presented with the following sequences?

Case 1: `"-2147483647"`

Case 2: `"-2147483648"`

Case 3: `"-2147483649"`

Interpretation:

The relevant citations are the ones supplied by you from § 4.10.1.6:

> If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.
>
> If the correct value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the macro `ERANGE` is stored in `errno`.

The committee believes that there is only one sensible interpretation of a subject sequence with a minus sign: If the subject sequence (neglecting the possible minus sign) is outside the range [0, `ULONG_MAX`], then the range error is reported. Otherwise, the value is negated (as an `unsigned long int`).

The answers to your numeric questions are:

Case 1: 2147483649

Case 2: 2147483648

Case 3: 2147483647

Request for Interpretation # 7 (X3J11 Doc. No. 90-043)

Question 1:

Are declarations of the form *struct-or-union identifier* `;` permitted after the *identifier* tag has already been declared? Here are some examples of the problem:

```
/*1*/  struct  s;
/*2*/  struct  s;
/*3*/  struct  s  {int  a;};
/*4*/  struct  s;

/*5*/  struct  t  {int  a;};
/*6*/  struct  t;
```

Standard § 3.5 says ‘‘A declaration shall declare at least a declarator, a tag, or the members of an enumeration.’’ In this sense, does `/*2*/` also declare the tag `s`? If so, then surely all of the above lines are conforming. But if not, then in what sense does `/*3*/` declare a tag and thus satisfy § 3.5’s constraint?

The example at the end of § 3.5.2.3 says ‘‘To eliminate this context sensitivity, the otherwise vacuous declaration `struct s2;` may be inserted ...’’. This seems to imply that `/*2*/`, `/*4*/`, and `/*6*/` are not conforming, because they are vacuous. But how can this be reconciled with the above argument?

Interpretation:

The declaration

```
struct  s;
```

declares the tag `s`. It need not be the first or only declaration of the tag `s` within a given scope to qualify as a declaration of `s`, just as

```
int  i;
```

declares `i` however often it is repeated. The applicable constraint is in § 3.5: ‘‘A declaration shall declare at least a declarator, a tag, or the members of an enumeration.’’ Clearly,

```
struct  s;
```

declares the tag `s`.

§ 3.5.2.3, in the examples, characterizes a declaration of this form as ‘‘otherwise vacuous’’. The words ‘‘otherwise vacuous’’ are an editorial comment and could be safely omitted without changing the meaning of the sentence. These words mean ‘‘other than declaring `s2` to be an (incomplete) struct type’’, and should not be read as saying that the declaration fails to declare the tag.

We believe that this interpretation is consistent with the intent of the committee, and that a reasonable reading of the standard supports this interpretation.

Request for Interpretation # 8 (X3J11 Doc. No. 90-021)

Question 1:

Could you tell me if it is legitimate for a conforming C compiler to perform what's commonly referred to as dead-store elimination for the first assignment in the following code fragment:

```
auto  int  flag;/*  non-volatile  */
...
flag  =  1;
flag  =  f();
```

If it is valid to do so, then consider

```
auto  int  flag;/*  non-volatile  */
if  (setjmp(buf))
     {
       if  (flag  ==  1)  ...
     }
flag  =  1;
flag  =  f();
```

where function `f` invokes `longjmp`. Is the result of the relational expression defined? A solution might be to define `flag` as `volatile`, but `flag` is *not* really volatile, and the programmer may not wish to degrade all references to `flag` nor to locate all such possible `flag`s and lie about their volatility.

A related issue is that in many existing applications, users have coded `setjmp`-like mechanisms based on a particular operational environment. The functions do not have the name ''`setjmp`'', but essentially establish an externally accessible entry point within the containing function. Sometimes, pointers are set to reference such functions, even though the standard precludes this from being done with `setjmp` itself since it is allowable that it only be provided as a macro.

There are a number of additional optimizations which must be inhibited across the actual invocation of `setjmp`, or a `setjmp`-like function. Always avoiding these optimizations as well as the dead-store elimination shown in the example may make the program safe for non-local jumps, but unnecessarily penalizes programs that don't use `setjmp`. To circumvent this problem, some implementors have defined a pragma which is included in `setjmp.h` to identify ''`setjmp`'' as having the property of establishing an externally accessible entry, *i.e.*, defining an otherwise non-obvious point of control flow. Other implementations have hard-coded tests for the name ''`setjmp`''.

… would you please respond to the question regarding the legitimacy of the optimization in the first example?

Interpretation:

The relevant citation is § 4.6.2.1

> All accessible objects have values as of the time `longjmp` was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding `setjmp` macro that do not have volatile-qualified type and have been changed between the `setjmp` invocation and `longjmp` call are indeterminate.

In response to your question about the effect on optimizations of `setjmp`: Yes, it is legitimate for a compiler to perform optimizations that eliminate dead stores to local, non-volatile, automatic variables when `setjmp` is used. § 4.6.2.1 makes the values of all such variables indeterminate after the `longjmp` is called. This grants a compiler the liberty to perform dead-store elimination as well as several other optimizations.

Question 2:

What is happening is that, since the standard has not provided a mechanism to describe a very recognizable and very important property of a function, such mechanisms are by necessity being provided in non-standard ways. My understanding is that a pragma should never be required for a program to execute correctly as defined by the standard.

The existing situation serves to reduce portability of C programs. We believe the committee should address this problem and would like to offer a suggestion which seems rather attractive.

Currently, defining an object as `volatile` indicates to the compiler that its contents may be altered in ways not under control of the implementation. This is meaningless with function declarations since a function doesn't have alterable contents (*i.e.*, is not an lvalue). Instead, it may be possible to utilize this otherwise syntactic no-op by defining a ''volatile function'' to be one whose return may not necessarily occur sequentially at the point of the invocation, but possibly at some other point where the state of the calling program is unknown. In other words, invocation of such a function results in the state of the program becoming volatile.

Now, I admit that this is not a perfectly ''clean'' extrapolation of the use of the type qualifier `volatile`, but it is rather compelling, having the following advantages:

1) It solves the described problem in a general way that can be used with functions not necessarily named ''`setjmp`''. Implementations defining `setjmp` as a function in `setjmp.h` would simply declare

```
int  volatile  setjmp(jmp_buf  env);
```

2) It utilizes an existing keyword and gives meaning to its use in a context which would be otherwise meaningless.

3) It is consistent with the type specifier syntax to distinguish between volatile pointers and pointers to volatile objects. For example,

```
int  volatile  setjmp();
```

defines `setjmp` to be a volatile function (*i.e.*, a function whose invocation must inhibit certain optimizations).

```
int  volatile  (*maybe_setjmp_ptr)();
```

defines a pointer to such a function, while

```
int  (*mustnotbe_setjmp_ptr)();
```

defines a pointer to a normal function.

```
int  (*  volatile  vol_mustnotbe_setjmp_ptr)();
```

defines a volatile pointer to a normal function.

```
int  volatile  (*  volatile  vol_maybe_setjmp_ptr)();
```

defines a volatile pointer to a volatile function, and so on ...

4) Type consistency rules are already in place and make sense. For example,

```
maybe_setjmp_ptr  =  mustnotbe_setjmp_ptr;
```

is okay with no type-checking violation, whereas

```
mustnotbe_setjmp_ptr  =  maybe_setjmp_ptr;
```

is diagnosed. It would require casting such as

```
mustnotbe_setjmp_ptr  =  (int  (*)())maybe_setjmp_ptr;
```

5) Since no new syntax or keywords are required, the impact of this change is very small to both the document defining the standard and to compilers which support it.

If there is enough committee interest in this sort of solution, I would be glad to draft a formal proposal.

Interpretation:

In response to your suggestion on volatile-qualified function return types: This is not a request for an interpretation; this is a request for changes to the standard. Changes cannot be considered until the next review of the language. The committee reasserts that the current semantics for type qualifiers as they appear in the standard are as intended.

Request for Interpretation # 9 (X3J11 Doc. No. 90-023)

Question 1: Use of typedef names in parameter declarations

A syntactic ambiguity exists in the draft proposed C standard for which there appears to be no semantic disambiguation. A sequence of examples should explain the ambiguity. This matter needs interpretation by the X3J11 committee.

For these examples, let `T` be declaration specifiers which contain at least one type specifier, to satisfy the semantics from § 3.5.6:

> If the identifier is redeclared in an inner scope ..., the type specifiers shall not be omitted in the inner declaration.

Let `U` be an identifier which is a typedef name at outer scope and which has not (yet) been redeclared at current scope. A caret indicates the position of each abstract declarator. Consider this declaration:

declaration-specifiers *direct-declarator* `(T^(U));`

Here `U` is the type of the single parameter to a function returning type `T`, due to a requirement from § 3.5.4.3:

> In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

Consider this declaration:

declaration-specifiers *direct-declarator* `(T^(U^(`*parameter-type-list*`)));`

In this example, `U` could be the type returned by a function which takes *parameter-type-list*. This in turn would be the single parameter to a function returning type `T`.

Alternatively, `U` could be a redundantly parenthesized name of a function which takes *parameter-type-list* and returns type `T`.

Given the spirit of the requirement from § 3.5.4.3, the former interpretation seems to be that intended by X3J11. If so, the requirement may be changed to something similar to:

> In a parameter declaration, a direct declarator which redeclares a typedef name shall not be redundantly parenthesized.

Of course, parentheses must not be disallowed entirely. ...

[The original had more, but this will suffice.]

Interpretation:

It is widely recognized that the context-free syntax of C contains several instances where an identifier could be taken as either a typedef name or as an ordinary identifier. The following general principle has guided the description of such cases:

> Whenever a typedef name could be taken as such in a declaration, it is so taken.

The specific citations of this intent are in § 3.5.4.3 Semantics, paragraph 3:

> In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

and in § 3.5.6 Semantics, paragraph 1:

> If the identifier is redeclared in an inner scope or is declared as a member of a structure or union in the same or an inner scope, the type specifiers shall not be omitted in the inner declaration.

The committee acknowledges that the Request for Interpretation has identified another such context, to be

disambiguated by the same general principle.

Request for Interpretation # 10 (X3J11 Doc. No. 90-044)

Question 1:

Consider:

```
typedef  int  table[];  /*  line 1  */
table  one  =  {1};     /*  line 2  */
table  two  =  {1,  2}; /*  line 3  */
```

First, is the typedef to an incomplete type legal? I can't find a prohibition in the standard. But an incomplete type is completed by a later definition, such as line 2, so what is the status of line 3?

The type, of which `table` is only a synonym, can't be completed by line 2 if it is to be used in line 3. And what is `sizeof(table)`? What old C compilers seem to do is treat the typedef as some sort of textual equivalent, which is clearly wrong.

Interpretation:

A typedef of an incomplete type is permitted.

Regarding objects `one` and `two`, refer to the standard § 3.1.2.5 page 25, lines 8-9: ''An array of unknown size is an incomplete type. It is completed, *for an identifier of that type*, by specifying the size in a later declaration …'' [emphasis added]. The types of objects `one` and `two` are completed but the type `table` itself is *never* completed. Hence, `sizeof(table)` is not permitted.

An example very similar to that submitted is shown in § 3.5.7 on page 75, lines 16-23.

Request for Interpretation # 11 (X3J11 Doc. No. 90-008)

Question 1: Merging of declarations for linked identifier

When more than one declaration is present in a program for an externally-linked identifier, exactly when do the declared types get formed into a composite type?

Certainly, if two declarations have file scope, then after the second, the effective type for semantic analysis is the composite type of the two declarations (§ 3.1.2.6, page 26, lines 19-20). However, if one declaration is in an inner scope and one is in an outer scope, are their types formed into a composite type?

In particular, consider the code:

```
{
extern  int  i[];
{
        /*  a different declaration of the same object  */
        extern  int  i[10];
}
/*  Is the following legal?  That is, does the outer declaration
      inherit any information from the inner one?  */
sizeof  (i);
}
```

Similar situations can be constructed with internally linked identifiers. For instance:

```
/*  File scope  */
static  int  i[];

main()
{
/*  a different declaration of the same object  */
extern  int  i[10];
}

/*  Is the following legal?  That is, does the outer declaration
      inherit any information from the inner one?  */
int  j  =  sizeof  (i);
```

Further variants of this question can be asked:

```
{
extern  int  i[10];
{
        /*  a different declaration of the same object  */
        extern  int  i[];

        /*  Is the following legal?  That is, does the inner declaration
              inherit any information from the outer one?  */
        sizeof  (i);
}
}
```

Interpretation:

For all three examples above, the following interpretation applies. The first relevant section in the standard is § 3.1.2.6 (‘‘Compatible Type and Composite Type’’):

> For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

The X3J11 committee unanimously agreed that since the two declarations of `i` are in different scopes, no composite type is created. Therefore, this usage of the `sizeof` operator must be diagnosed, because it violates the

following constraint in § 3.3.3.4 (‘‘The `sizeof` Operator’’):

> The `sizeof` operator shall not be applied to an expression that has function type or an incomplete type, …

Question 2: Interpretation of `extern`

Consider the code:

```
/*  File scope  */
static  int  i;          /*  declaration 1  */

main()
{
extern  int  i;          /*  declaration 2  */
{
       extern  int  i;   /*  declaration 3  */
}
}
```

A literal reading of § 3.1.2.2 says that declarations 1 and 2 have internal linkage, but that declaration 3 has external linkage (since declaration 1 is not visible, being hidden by declaration 2). (This combination of internal and external linkage is illegal by § 3.1.2.2, page 22, line 27.)

Is this what is intended?

Interpretation:

§ 3.1.2.2 (‘‘Linkages of Identifiers’’) contains the following two excerpts:

> If the declaration of an identifier for an object or a function contains the storage class specifier `extern`, the identifier has the same linkage as any visible declaration of the identifier with file scope.
>
> If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

The X3J11 committee unanimously agreed that, since the lexically third declaration of `i` does not link with a file scope declaration (because declaration 1 is currently not visible), there exists an identifier with both internal and external linkage, which is an error. This is neither a syntax nor a constraint error. Therefore, no diagnostic is required.

Question 3: Initialization of tentative definitions

If the file scope declaration

```
int  i[10];
```

appears in a translation unit, § 3.7.2 suggests that it is implicitly initialized as if

```
int  i[10]  =  0;
```

appears at the end of the translation unit. However, this initializer is invalid, since § 3.5.7 prescribes that the initializer for any object of array type must be brace-enclosed. We believe that the intention of § 3.7.2 is that this declaration has an implicit initializer of

```
int  i[10]  =  {0};
```

Is this true?

Interpretation:

§ 3.7.2 (‘‘External Object Definitions’’) contains the following excerpt:

> If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

The X3J11 committee unanimously agreed that this statement describes an effect and not a literal token sequence. Therefore, this example does not contain an error.

Question 4: Tentative definition of externally-linked object with incomplete type

If one writes the file-scope declaration

```
int  i[];
```

then § 3.7.2 suggests that at the end of the translation unit the implicit declaration

```
int  i[]  =  {0};
```

or equivalently

```
int  i[1]  =  {0};
```

appears. This seems peculiar, since § 3.7.2, (page 84, lines 35-36) specifically forbids this case for internally linked identifiers.

Is this what is intended?

Interpretation:

§ 3.7.2 (‘‘External Object Definitions’’) contains the following excerpt:

> If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

The X3J11 committee unanimously agreed that the dimension is *one* at the end of the translation unit (but not before).

Request for Interpretation # 12 (X3J11 Doc. No. 90-046)

Question 1: Bug in Standard C

I was asked a question about the validity of various expressions. Among the list there was the following:

```
void  *p;  &*p;
```

After doing a quick pass through the standard, I found nothing that disallowed such. Moreover, back in September 1987's meeting (I didn't just recall the date ... it took a while to find when it occurred), I distinctly remember a Committee discussion that involved the validity of the expression above. It was as a result of this discussion and vote that the draft was changed to allow the above.

Anyway, I wrote back that the expression was valid. This was eventually followed by a letter from Dennis [Ritchie] pointing out the mistake I made. As it turns out, the definition of lvalue makes at least the unary & part of the above a constraint violation. (As Bill [Plauger] would say, ''I know what the standard was *supposed* to specify''.)

This would be just another ''oops, well I guess I can live with it'' surprise in the standard, except that it turns out that unary & of a `void` type is useful! What it provides is a construction that gives C a notion of an address symbol. You are familiar with the symbols that are created by the UNIX® linker: `etext`, `edata`, and `end`, which designate special addresses within the `a.out`'s address space. Of these, the last is most useful (it gives the beginning of the dynamically allocated data space). However, the *type* for these symbols was always pretty fuzzy. But, consider a declaration of `end` as

```
extern  void  end;
```

What this gives is a name that only has an address — exactly what these symbols do, and nothing more. They can only be used in C as the operand of unary &, and the address must be converted to something else (say, `char *`) even to do address calculation, making the special nature of the symbol clearly evident.

What I'd like is a vote of the interpretations group that notes that the intent of the Committee was that ''`void *p; &*p;`'' was supposed to be valid, even though a conforming implementation must diagnose the expression. This means that I can continue to suggest the ''`extern void`'' approach to address symbols in C.

P.S. The following is my reply to Dennis's mail that pointed out the error with my original interpretation. The indented parts are from Dennis's mail.

> I don't agree with Dave P's answer about ''`void *vp; &*vp;`''. There is not a constraint on `*`, but the § 3.3.3.2 semantics say, ''... if it [the operand of `*`] points to an object, the result is an lvalue designating the object.'' Does `vp` point to an object? An object is ''a region of data storage ... the contents of which can represent values'' (§ 1.6). Dicey at best.

I took some time looking into my records of the Committee's thoughts on this very issue. Back in '87, based on a proposal by Plauger, the Committee voted 27 to 3 that ''`*(void *)`'' was not to be an error. This was when the unary * constraint was simplified to the current form. Since `void` is a special instance of an incomplete object type, it can be thought of as pointing at an object whose size we do not know, but I agree that the argument is strained. I would still recommend that the compiler not produce a hard error in this situation.

> Moreover, the operand of & must be an lvalue, and `*vp` is certainly not an lvalue (§ 3.2.2.1): ''An *lvalue* is an expression (with an object type or an incomplete type other than `void`) ...''

Oops. In this case, I completely agree with Dennis: the standard does say that unary & should not be applied to an

---

® UNIX is a registered trademark of UNIX System Laboratories, Inc., in the United States and other countries.

expression with type `void` since such cannot be an lvalue. Unfortunately, this means that the standard is ‘‘broken’’, at least according to the Committee’s decisions. One of the major arguments presented as part of the September 1987 meeting for allowing ‘‘`*(void *)`’’ was that it could then be immediately used as the operand of unary `&`!

Therefore, I can state that back in 1987, the Committee’s intent was that the examples you gave were valid Standard C, but that the standard as written does not allow the second half of the construction for `void`! Nevertheless, I’d still suggest allowing the code to successfully compile, with at most a warning.

Interpretation:

The relevant citations are § 3.3.3.2 (page 44, lines 36-38)

> The operand of the unary `&` operator shall be either a function designator or an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier.

and the one supplied by you from § 3.2.2.1 (page 37, lines 3-4)

> An *lvalue* is an expression (with an object type or an incomplete type other than `void`) that designates an object.

Given the following declaration:

```
void  *p;
```

the expression `&*p` is invalid. This is because `*p` is of type `void` and so is not an lvalue, as discussed in the quote from § 3.2.2.1 above. Therefore, as discussed in the quote from § 3.3.3.2 above, the operand of the `&` operator in the expression `&*p` is invalid because it is neither a function designator nor an lvalue.

This is a constraint violation and the translator must issue a diagnostic message.

The desired effect can be obtained by using the declaration

```
extern  const  void  end;
```

(where `end` denotes an object of unknown size) since `const void` type is not `void` type and thus `&end` does not violate the constraint in § 3.3.3.2.

Informal discussion

The following notes are an unofficial summary of discussion. They do not reflect X3J11 decisions, but may be helpful nonetheless.

Footnote 6 (page 7), which is not part of the standard, provides a suggestion for implementors who may wish to assign a meaning to the above expression. It says ‘‘An implementation may also successfully translate an invalid program.’’ Therefore, as long as a diagnostic message is issued, a translator may assign a meaning to the expression `&*p` discussed above. Conforming programs shall not use this expression, however.

Request for Interpretation # 13 (X3J11 Doc. No. 90-047)

Question 1: Compatible and composite function types

A fix to both problems Mr. Jones raises in X3J11 Document Number 90-006 is: In § 3.5.4.3 on page 69, lines 23-25, change the two occurrences of

> its type for these comparisons

to

> its type for compatibility comparisons, and for determining a composite type

This change makes the sentences pretty awkward, but I think they remain readable.

This change makes all three of Mr. Jones's declarations compatible:

```
int  f(int  a[4]);
int  f(int  a[5]);
int  f(int  *a);
```

This should be the case; it is consistent with the base document's idea of ''rewriting'' the parameter type from array to pointer.

Interpretation:

The parenthetical sentence in § 3.5.4.3 on page 69, lines 22-24 applies to the entire paragraph starting at line 12. Therefore the types of the two functions

```
int  f(int  a[5]);
```

and

```
int  f(int  *a);
```

are compatible and no wording change is needed.

The author is correct that the standard's rules for determination of a resultant composite type are flawed because [§ 3.1.2.6, page 26, line 16] there is no way to construct the composite type of `int [5]` and `int *` — they are not compatible.

Besides § 3.5.4.3 page 69, line 22, § 3.7.1 page 83, line 24 is further evidence that the committee meant that `int [5]` be converted to `int *`. It was specified for definitions here; it probably should have been placed in declarations instead. We conclude it was our only reasonable intent that `int [5]` be treated as `int *` for the purposes of determining a composite type.

Question 2: ''Compatible'' not defined for recursive types

The term ''compatible'' is not completely defined. Consider the following two file-scope declarations in *separate* translation units:

```
extern  struct  a  {  struct  a  *p;  }  x;
struct  a  {  struct  a  *p;  }  x;
```

Are these two declarations of `x` compatible? Obviously they should be, but § 3.1.2.6 does not say so.

Because § 3.1.2.6 does not say how to terminate the recursion in testing for compatibility of two recursive types, either interpretation is possible. In other words, it is consistent with the rules in § 3.1.2.6 to decide that the two declarations are compatible; but it is also consistent to decide that they are incompatible.

We can solve the problem roughly as follows: append the following draft sentence to the first paragraph of § 3.1.2.6 (page 26, line 8):

> If two types declared in separate translation units admit the possibility of being either compatible or incompatible, the two types shall be compatible.[footnote]
>
> [footnote] This case occurs with recursive types.

This sentence is not satisfactory; perhaps another committee member can state this rule better.

Interpretation:

We agree that the standard ‘‘loops’’. Our intent, and we feel the only reasonable solution, is that the recursion stops and the two types are regarded as compatible.

Question 3: Composite type of `enum` *vs.* integer not defined

There is one case where two types are compatible, but their composite type is not defined. To fix this problem, in § 3.1.2.6 insert after page 26, line 17:

- If one type is an enumeration and the other is an integer type, the composite type is the enumeration.

There may be other cases where ‘‘compatible’’ is not defined. I made a cursory search and did not find any.

Interpretation:

The issue is that in

```
enum  {r,w,b}  x;
```

and

```
some-int-type   x;
```

where *some-int-type* happens to be the type that by § 3.5.2.2, page 62, line 40 is compatible with the type of the `enum`, what is the resultant composite type?

§ 3.1.2.6 on page 26, lines 11-12 says ‘‘*a* type that … satisfies the following conditions’’ [added emphasis on ‘‘*a*’’]. The composite type of two compatible types is not necessarily unique. In this case both the `enum` type and the *some-int-type* satisfy the definition of ‘‘composite’’ type. This refutes Kendall’s claim that the ‘‘composite type is not defined’’; the point is that the standard does not guarantee a *unique* composite type.

Question 4: When a structure is incomplete

Reference § 3.5.2.3, page 63, lines 25-28:

> If a type specifier of the form
>
> *struct-or-union identifier*
>
> occurs prior to the declaration that defines the content, the structure or union is an incomplete type.

In the following example, neither the second nor the third occurrence of `struct foo` seem adequately covered by this sentence:

```
struct  foo  {
        struct  foo  *p;
}  a[sizeof  (struct  foo)];
```

In the second occurrence `foo` is incomplete, but since the occurrence is within ‘‘the declaration that defines the content’’, it cannot be said to be ‘‘prior’’ that declaration. In the third occurrence `foo` is complete, but again, the

occurrence is within the declaration.

To fix the problem, change the phrase ‘‘prior to the declaration’’ to

> prior to the end of the *struct-declaration-list* or *enumerator-list*

Interpretation:

For the example

```
struct  s  {
         struct  s  *p;
         }          ^ point 1
           X  [sizeof(struct  s)];
                                  ^ point 2
```

the issue is whether the language in § 3.5.2.3 on page 63, line 27 specifies that *points 1* and *2* are ‘‘prior to the declaration’’ or not. The implication is that they denote an incomplete type if and only if they are prior.

The committee felt that the wording in the standard did not accurately reflect the intent, because ‘‘prior to the declaration’’ does not cover the range of text within the declaration. However, we unanimously agreed that *point 1* was incomplete and *point 2* complete. Perhaps we should have said ‘‘occurs prior to the complete definition of the content’’. An alternative wording would be ‘‘occurs prior to the } following the *struct-declaration-list* that defines the content’’. Either wording captures what the committee has always meant here, and so we interpret the wording in the standard to be that meaning.

Question 5: Enumeration tag anomaly

Consider the following (bizarre) example:

```
enum  strange1  {
        a  =  sizeof  (enum  strange1)   /*  line [2]  */
};
enum  strange2  {
        b  =  sizeof  (enum  strange2  *)        /*  line [5]  */
};
```

The respective tags are visible on lines [2] and [5] (according to § 3.1.2.1, page 21, lines 39-40), but there is no rule in § 3.5.2.3 Semantics (page 63) that governs their meaning on lines [2] and [5]. Footnote 62 on page 63 seems to be written without taking this case into account.

The first declaration must be illegal. The second declaration should be illegal for simplicity.

Perhaps these declarations are already illegal, since no rule gives them a meaning. To clarify matters, I suggest in § 3.5.2.3 appending to page 63, line 35:

> A type specifier of the form
>
> `enum` *identifier*
>
> shall not occur prior to the end of the *enumerator-list* that defines the content.

If this sentence is not appended, something like it should appear as a footnote.

Interpretation:

For the example

```
enum e { a = sizeof(enum e); };
```

the relevant citations are § 3.1.2.1 starting on page 21, line 39, indicating that the scope of the first `e` begins at the `{`, and § 3.5.2.2, page 62, line 20, which attributes meaning to a later `enum e` *only if* this use appears in a *subsequent* declaration. By subsequent we mean ‘‘after the `}`’’. Because in this case the second `enum e` is not in a subsequent declaration, and no other wording in the standard addresses the meaning, the standard has left this example in the category of undefined behavior.

Request for Interpretation # 14 (X3J11 Doc. No. 90-049)

Question 1: X/Open Reference Number KRT3.159.1

There are conflicting descriptions of the `setjmp()` interface in ANS X3.159-1989. In § 4.6 on page 119 line 8, it is stated that ''It is unspecified whether `setjmp` is a macro or an identifier declared with external linkage.'' Throughout the rest of the standard, however, it is referred to as ''the `setjmp` macro''; in addition, the rationale document states that `setjmp` must be implemented as a macro. Please clarify whether `setjmp` must be implemented as a macro, or may be a function as well as a macro, or may just be a function.

Interpretation:

The standard states that `setjmp` can be either a macro or a function. It is referred to as ''the `setjmp` macro'' just to avoid longwindedness. The rationale document is incorrect in saying that it must be a macro.

Question 2: X/Open Reference Number KRT3.159.2

Standard § 4.9.6.2 (''The `fscanf` Function'') states:

> If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current input directive have been read (other than leading white space, where permitted), execution of the current directive terminates with input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

How should an implementation behave when end-of-file terminates an input stream that satisfies all conversion specifications that consume input but there is a remaining specification request that consumes no input (*e.g.* `%n`)? Should the non-input-consuming directive be evaluated or terminated with an input failure as described above?

Interpretation:

The intention was to *exclude* `%n` from the phrase ''… execution of the following directive (if any) is terminated with an input failure''.

Evidence:

1. The entire ''otherwise'' clause above was unnecessary! Without it, this issue would be clear.
2. According to § 4.9.6.2, page 138, line 1, `%n` consumes no input. § 4.9.6.2, page 136, lines 20-21 describe input failure. Therefore, `%n` should never get an input failure.
3. § 4.9.6.2 on page 138, lines 17-18 says: ''The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.'' This does not guarantee that `%n` can do this, but it does imply it.

Request for Interpretation # 15 (X3J11 Doc. No. 90-051)

Question 1:

This question concerns the promoted type of plain `int` bit-fields with length equal to the size of an object of type `int`. I am interested in implementations which have chosen not to regard the high-order bit as a sign bit.

The question is: What is the promoted type of such an object?

§ 3.5.2.1 states

> A bit-field shall have a type that is ... `int`, `unsigned int`, or `signed int`.

The intent of this, I believe, is that the type of a plain `int` bit-field is `int.`

§ 3.2.1.1 states

> A `char`, a `short int`, or an `int` bit-field, or their signed or unsigned varieties, ... may be used in an expression wherever an `int` or `unsigned int` may be used. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise it is converted to an `unsigned int`. ...
>
> The integral promotions preserve value including sign.

Tracing this through, then, the type of any promoted plain `int` bit-field is `int`, since `int` can hold all the values of the original type, which is `int`. However, not all values of the bit-field, which may be regarded as non-negative, can be represented by an `int`. By value-preserving promotion rules, I would expect the type of the promoted bit-field to be `unsigned int`.

Can you clarify this?

Interpretation:

Bit-fields that are being treated as unsigned will promote according to the same rules as other unsigned types: if the width is less than `int`, and `int` can hold all the values, then the promotion is to `int`. Otherwise, promotion is to `unsigned int`.

§ 3.2.1.1 says ‘‘The integral promotions preserve value including sign. ...’’ Sign of the value is preserved, not signedness of the type.

Request for Interpretation # 16 (X3J11 Doc. No. 90-052)

Question 1:

I can find no prohibition of the following translation unit:

```
struct  foo  x;
struct  foo  {  int  i;  };
```

What I was looking for, but didn't find, was a statement that an implicitly initialized declaration of an object with static storage duration must have object type.

Is this translation unit legal?

Interpretation:

The translation unit cited is legal. It falls into the same category of construct as

```
int  array[];
...
int  array[17];
```

Objects may be declared without knowing their size. However, the standard is clear in what cases such an object may or may not used, prior to the actual definition of the object.

Question 2:

This one is relevant only for hardware on which either null pointer or floating point zero is *not* represented as all zero bits.

Consider this sentence in § 3.5.7 (starting on page 72, line 40):

> If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

This implies that you cannot implicitly initialize a union object that could contain overlapping members with different representations for zero/null pointer. For example, given this translation unit:

```
union  {  char  *p;  int  i;  }  x;
```

If the null pointer is represented as, say, `0x80000000`, then there is no way to implicitly initialize this object. Either the `p` member contains the null pointer, or the `i` member contains 0, but not both. So the behavior of this translation unit is undefined.

This is a bad state of affairs. I assume it was not the committee's intention to prohibit a large class of implicitly initialized unions; this would render a great deal of existing code nonconforming.

The right thing — although I can find no support for this idea in the draft — is to implicitly initialize only the first member of a union, by analogy with explicit initialization. Here is a proposed new sentence; perhaps it can be saved for the next time we make a C standard. (This sentence also tries to get around the difficulty of the old ''as if … assigned'' language in dealing with `const` items; Dave Prosser tipped me off there.)

If an object that has static storage duration is not initialized explicitly, it is initialized implicitly according to these rules:

- if it is a scalar with pointer type, it is initialized implicitly to a null pointer constant;
- if it is a scalar with non-pointer type, it is initialized implicitly to zero;
- if it is an aggregate, every member is initialized (recursively) according to these rules;
- if it is a union, the first member is initialized (recursively) according to these rules.

Interpretation:

The committee agreed that a clear statement of the effect of implicit initialization of unions is missing. However, it was the consensus of the committee that the intent was clearly that it should behave as if the first member of the union was initialized to 0. The conversion to any non-zero internal representation of 0 for the particular member type is covered by the wording in § 3.5.7, page 73, lines 3-5, which indicate that the same conversions as for simple assignment apply.

Request for Interpretation # 17 (X3J11 Doc. No. 90-056)

Question 1: New-line in preprocessor directives

§ 2.1.1.2, page 6, line 37 says: ‘‘Preprocessing directives are executed and macro invocations are expanded.’’

§ 3.8, page 87, lines 2-5 say: ‘‘A preprocessing directive … and is ended by the next new-line character.’’

§ 3.8.3, page 90, lines 38-39 say: ‘‘Within the sequence of preprocessing tokens … new-line is considered a normal white-space character.’’

These three statements are not sufficient to categorize the following:

```
#define  f(a,b)  a+b
#if  f(1,
              2)
...
```

It should be defined whether the preprocessing directive rule or macro expansion wins, *i.e.* is this code fragment legal or illegal?

In translation phase 4 ‘‘preprocessing directives are executed and macro invocations expanded’’.

Now do macro invocations get done first, followed by preprocessor directives? Does the macro expander need to know that what it is expanding forms a preprocessing directive?

§ 3.8, page 87, lines 2-5 suggest that the preprocessor directive is examined to look for the new-line character. But how is it examined? Obviously phases 1-3 happen during this examination. So why shouldn’t part of phase 4?

Proposed resolution
Insert in § 3.8 on page 87 between lines 16-17 either of:*

1) Any new-line characters occurring within the argument list of a function macro invocation do not end the preprocessing directive.

2) Any new-line character occurring within the argument list of a function macro invocation ends the preprocessing directive.

The majority of those expressing an opinion indicated that they did not mind which reading was selected by the interpretation committee.

Interpretation:

We believe that § 3.8 page 87, lines 2-5 clearly require that the first new-line character terminates a preprocessing directive. This termination is not altered by § 3.8.3 page 90, lines 38-40, which state that ‘‘new-line is considered a normal white-space character’’ within an invocation of a function-like macro, since the new-line character is not changed by this statement. Thus,

```
#define  f(a,b)  a+b
#if  f(1,
              2)
```

has a clear interpretation. The `#if` preprocessing directive contains a partial macro invocation, and is therefore ill-formed. Nothing in the standard gives the translator license to read beyond the new-line character to complete the macro invocation.

---

* The *proposed resolutions* in this Request for Interpretation were proposed for a normative addendum to the International Standard ISO/IEC 9899:1990.

Question 2: Behavior if no function called `main` exists

According to § 2.1.2.2.1, page 7, it is implicitly undefined behavior if the executable does not contain a function called `main`.

It ought to be explicitly undefined if no function called `main` exists in the executable.

Proposed resolution
Add to § 2.1.2.2 on page 7 between lines 33-34:

> If a function called `main` is not present at program startup the behavior is undefined.

Interpretation:

You are correct that it is implicitly undefined behavior if the executable does not contain a function called `main`. This was a conscious decision of the X3J11 committee.

There are many places in the standard that leave behavior implicitly undefined. The X3J11 committee chose as a style for the standard not to enumerate these places as explicitly undefined behavior. Rather, § 1.6 on page 3, lines 44-46 explicitly allows for implicitly undefined behavior and explicitly gives implicitly undefined behavior equal status with other forms of undefined behavior.

Question 3: Precedence of behaviors

Refer to § 3.1.2.6, page 26, lines 9-10 and § 3.5, page 58, lines 20-21. The constructs covered by these sentences overlap. The latter is a constraint while the former is undefined behavior. In the overlapping case who wins?

Proposed resolution
Add either of the following paragraphs:

1) If a construct violates a constraint and is also specified as having undefined or implementation defined behavior the constraint takes precedence.

2) If a construct violates a constraint and is also specified as having undefined or implementation defined behavior the constraint does not take precedence.

Interpretation:

When a construct violates a constraint, § 2.1.1.3 on page 7, lines 15-17 requires a diagnostic, with no exceptions described. That a particular construct might also be in the category of undefined behavior does not release a conforming implementation from issuing a diagnostic.

Therefore, we believe that the current wording in the standard is clear and does not require the editorial addition that you propose.

Question 4: Mapping of escape sequences

Refer to § 3.1.3.4, page 30, line 12 and line 16. Are these values the values in the source or execution character set?

When § 3.1.3.4, page 30, line 24 says: ‘‘The value of an …’’, is this ‘‘value’’ the value in the source character set of the escape sequence or the value of the mapped escape sequence? I would have said that the ‘‘value’’ is the value in the execution environment since in the source environment `\x123` is a token.

It might be argued that characters in the source character set do not have values and thus no misinterpretation of ‘‘value’’ can occur. § 2.2.1, page 11, lines 25-26 refer to the value of a character in the source basic character set.

Proposed resolution
Amend § 3.1.3.4, page 30, line 12 to describe the value as being in the execution character set; change ‘‘The numerical value of the …’’ to ‘‘The numerical mapped value of the …’’. Similarly for lines 16-17.

Also, on line 24 change ‘‘The value of …’’ to ‘‘The mapped value of …’’.

Interpretation:

The values of octal or hexadecimal escape sequences are well defined and not mapped. For instance, the value of the constant `'\x12'` is always 18, while the value of the constant `'\34'` is always 28.

The mapping described in § 3.1.3.4 on page 29, lines 35-39 only applies to members of the source character set, of which octal and hexadecimal escape sequences clearly are not members.

Question 5: Example of value of character constants

Refer to § 3.1.3.4, page 30, lines 24-25 and page 31, lines 9-10. Both of these statements cannot be true.

a) If the constraint is violated, end of story. There is no implementation-defined value.

b) The implementation-defined behavior may be referring to the mapping of the escape sequence to the basic character set, in which case § 3.1.3.4, page 30, lines 24-25 should be changed to mention that it will violate a constraint if the mapped value is outside the range of representable values for the type `unsigned char`.

Proposed resolution
Change § 3.1.3.4, page 31, line 10 from ‘‘… and violates the above constraint’’ to ‘‘… and may violate the above constraint’’.

Interpretation:

The values of octal or hexadecimal escape sequences are well defined and not mapped. For instance, the value of the constant `'\x123'` has the value 291.

The mapping described in § 3.1.3.4 on page 29, lines 35-39 only applies to members of the source character set, of which octal and hexadecimal escape sequences clearly are not members.

The constraint in § 3.1.3.4 on page 30, lines 24-25 will be violated only if the implementation uses characters of eight bits.

The text of the example in § 3.1.3.4 on page 31, lines 8-10 is slightly opaque, but the parenthesized comment is meant to be subject to the words ‘‘Even if eight bits are used …’’ The value is implementation-defined only in that the implementation specifies how many bits are used for characters and whether type `char` is signed or not.

This example could be worded a little more clearly to indicate what is implementation-defined about the constant, and that it ‘‘violates the above constraint’’ only if eight bits are used for objects that have type `char`, but we believe that this interpretation is consistent with the intent of the committee, and that a reasonable reading of the standard supports this interpretation.

Question 6: `register` on aggregates

```
void  f(void)
{
register  union{int  i;}  v;
&v;            /*  Constraint error  */
&(v.i);    /*  Constraint error or undefined?  */
}
```

In § 3.3.3.2 on page 44, lines 37-38, in a constraint clause, it says ‘‘… and is not declared with the `register` storage-class specifier.’’ But in the above the field `i` is not declared with the `register` storage-class specifier.

Footnote 58, on page 59, states that ‘‘… the address of any part of an object declared with storage-class specifier `register` may not be computed …’’. Although the reference to this footnote is in a constraints clause I think that

it is still classed as undefined behavior.

Various people have tried to find clauses in the standard that tie the storage class of an aggregate to its members. I would not use the standard to show this point. Rather I would use simple logic to show that if an object has a given storage class then any of its constituent parts must have the same storage class. Also the use of storage classes on members is syntactically illegal.

The question is not whether such a construction is legal but the status of its illegality. Is it a constraint error or undefined behavior?

It might be argued that although `register` does not appear on the field `i`, its presence is still felt. I would point out that the standard does go to some pains to state that in the case of `const union{...}` the `const` does apply to the fields. The fact that there is no such wording for `register` implies that `register` does not follow the `const` rule.

Proposed resolution
Add to § 3.5.1 on page 59 between lines 18-19 either of the following:

1) A declaration of an aggregate with storage-class specifier `register` implicitly causes all of its members without the storage-class specifier `register` to be given the `register` storage-class specifier.

2) Members of an aggregate with the storage-class specifier `register` are not implicitly given the `register` storage class.

No comments received suggested that `&(v.i)` should be legal.

Interpretation:

The relevant citations are § 3.3.3.2

> The operand of the unary & operator shall be either a function designator or an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier.

and § 1.6

- Object — a region of data storage in the execution environment, the contents of which can represent values. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, …

and § 3.5.1, Footnote 58

> However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier `register` may not be computed, either explicitly (by use of the unary & operator as discussed in § 3.3.3.2) or implicitly …

In your example:

```
register  union  {int  i}  v;
&v.i;
```

you ask if the constraint in the standard that prohibits taking the address of `v` also applies to taking the address of `v.i`. The answer is yes.

The object that is the operand of & in `&v.i` is `v.i`. The member `v.i` is a part of the enclosing contiguous object `v` declared with the `register` storage-class specifier. Footnote 58 referenced above makes clear the committee's intention that the constraint in § 3.3.3.2 applies.

It is the committee's assertion that it is unreasonable to assume that the storage class of an object is not also the storage class of all parts of the object. Consider that the syntax of the language does not allow storage-class specifiers for parts of objects such as members. Given that the storage class in a declaration applies to all of the objects declared in the declaration, is it reasonable to assume that it does not equally apply to all subobjects of those objects? Consider the properties of an object determined by storage class. Many of those properties break down for the whole object if they are not also true for all parts of the object. For example, consider an object with

static storage duration whose parts only have automatic storage duration.

Question 7: Scope and uniqueness of `size_t`

§ 3.3.3.4 on page 46, lines 1-2 says: ‘‘… and its type (…) is `size_t` defined in the `<stddef.h>` header.’’ This line could be read as either of the following:

1) ‘‘… and its type is `size_t` which happens to be defined in `<stddef.h>`.’’
2) ‘‘… and its type is the `size_t` defined in `<stddef.h>`.’’

(It was probably intended as a helpful piece of information only.)

So what does the compiler do?

In 1) the compiler has to define a `size_t` in some outer scope. This definition does not make `size_t` visible, but gives a type to the return value of `sizeof`. Now if the programmer defines a typedef making `size_t` synonymous with `float` (say) then the compiler now has to use this new type. This interpretation does not require the programmer to include `<stddef.h>` in order to use `sizeof`.

In 2) the compiler picks up the type `size_t` from `<stddef.h>` (assuming that the user included this header). Should the compiler give a diagnostic if this header was not included and `sizeof` was used? A subsequent typedef for `size_t` does not affect the type of the result of `sizeof`.

These problems do not arise with `int` *et. al.* because they are keywords. Thus ‘‘`typedef float int`’’ would give a syntax error and need not be considered semantically.

According to § 3.3.3.4, page 46, `sizeof` has type `size_t`. What happens if the type of `size_t` does not match what the compiler thinks is the type of `sizeof`?

Proposed resolution
Change § 3.3.3.4 page 46, lines 1-2 from

> The value of the result is implementation-defined, and its type (an unsigned integral type) is `size_t` defined in the `<stddef.h>` header.

to

> The value of the result is implementation-defined, and its type is an unsigned integral type. If the standard header `<stddef.h>` has been included and the `typedef size_t` contained within it is not compatible with this unsigned type the behavior is undefined.

Arguments for/against
Comments received suggest that the following is the intended behavior.

At the file scope a compiler creates the following magic prototype:

```
a_nameless_type  sizeof  (any_type_allowed);
```

The `typedef size_t` found in `<stdlib.h>` should be compatible with `a_nameless_type`.

This interpretation removes all sorts of difficulties. The current wording does not reflect this intent.

The same trick could also be used for `ptrdiff_t`, but without the ability or neatness of defining a prototype.

Interpretation:

The relevant citations are § 3.3.3.4

> The value of the result is implementation-defined, and its type (an unsigned integral type) is `size_t` defined in the `<stddef.h>` header.

and § 4.1.5

> The types are …
>
> `size_t`
>
> which is the unsigned integral type of the result of the `sizeof` operator; …

These sections, both separately and together, define the relationship between the result type of `sizeof` and the type `size_t` defined in `<stddef.h>`. The result type of `sizeof` and the type `size_t` defined in `<stddef.h>` are an unsigned integral type, and `size_t` defined in `<stddef.h>` is identical to the result type of `sizeof`. To restate, in a conforming implementation, the result type of `sizeof` will be the same as the type of `size_t` defined in `<stddef.h>`.

Since these two types are the same, there need be no mechanism for a compiler to discover the type of `size_t` defined in `<stddef.h>`. A compiler's private knowledge of the result type of `sizeof` is as good as `<stddef.h>`'s private knowledge of the type of `size_t`.

Note that the result of `sizeof` has the same type as not just any `size_t`, but the `size_t` defined in `<stddef.h>`. This means that programmer definitions of `size_t` are ignored.

Question 8: Compatibility of pointer to `void` with storage class

Refer to § 3.3.9, page 50, lines 24-25. Do these lines make the following legal?

```
register  void  *p;
char  *q;
if  (p==q)    /*  legal?  */
…
```

The wording on line 25 ‘‘… version of `void`; or’’ does not talk about the ‘‘`void` type’’. This sentence could be taken as simply referring to the occurrence of a qualified or unqualified occurrence of `void`.

Should the wording on line 25 be changed to ‘‘… version of the type `void`; or’’ and thus cause the storage class to be ignored, or does the above example fall outside the scope of the constraint?

Proposed resolution
Change wording in § 3.3.9 on page 50, line 25

> … version of `void`; …

to

> … version of the `void` type; …

Interpretation:

The relevant citation is § 3.3.9

- one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`; or

The committee believes that the current wording of the standard is clear, and it is not changed in meaning by changing ‘‘version of `void`’’ in the quoted section to ‘‘version of the `void` type’’.

The standard uses the word ‘‘`void`’’ in two contexts: the keyword itself and the type that the keyword names. The context that the word is used in adequately distinguishes between the two. In the section quoted, which discusses type compatibility, a misreading of ‘‘`void`’’ as meaning the keyword quickly results in nonsense.

As to the qualification discussed in the quoted passage, it is type qualification, defined in § 3.5.3. The standard only uses the words ‘‘qualified’’ and ‘‘unqualified’’ when discussing type qualification and never uses them when discussing storage classes. Thus, storage classes have no place in the discussion of the quoted passage.

Question 9: Syntax of assignment expression

In § 3.3.16.1 on page 54, lines 31-32. there is a typo: ‘‘… of the assignment expression …’’ should be ‘‘… of the unary expression …’’

In § 3.3.16 on page 54, lines 3-5 we have

> *assignment-expression:*
>
> …
>
> *unary-expression assignment-operator assignment-expression*

Now the string ‘‘*assignment-expression*’’ occurs twice.

The use of ‘‘assignment expression’’ in § 3.3.16 on page 54, line 12 refers to the first occurrence (the one to the left of the colon).

We suggest changing the use of ‘‘assignment expression’’ in § 3.3.16.1 on page 54, line 32 in order to prevent confusion. The fact that any qualifier is kept actually makes more sense, since this qualifier has to take part in any constraint checking.

Interpretation:

The relevant citations are § 3.3.16.1

> In *simple assignment* (`=`), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

and § 3.3.16

> The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand.

You ask if the wording of § 3.3.16.1 should be changed from ‘‘of the assignment expression’’ to ‘‘of the unary expression’’. This change would be incorrect: it would change the result type of the expression by retaining any type qualifiers on the left operand.

The ‘‘assignment expression’’ in ‘‘of the assignment expression’’ on lines 31-32 is referring to the expression as a whole.* The type of the expression as a whole is defined in the above quoted passage from § 3.3.16.

* *I.e.,* it refers to line 3, not line 5.

Question 10: When is `sizeof` needed?

Refer to § 3.5.2.3, page 63, lines 28-29: When is the size of an incomplete structure needed? An interpreter may not need the size until run time, while some strictly typed memory architecture may not even allow pointers to structures of unknown size.

In § 3.5.2.3, Footnote 63 starts off as an example. The last sentence contains a ‘‘shall’’. Does a violation of this ‘‘shall’’ constitute undefined behavior?

Even though an interpreter may not need the size of a structure until run time its compiler still has to do some checking, *i.e.* an unexecuted statement may contain `sizeof` an incomplete type; even though the statement is unexecuted the constraint still has to be detected.

Proposed resolution
Add a sentence to Footnote 63 in § 3.5.2.3 on page 63

> Where the size of an object is needed is implementation-defined.

The above sentence explicitly specifies that behaviors other than the example may be chosen and that they should be documented.

Interpretation:

Whether the language processor is an interpreter or a true compiler does not affect the language rules about when the size of an object is needed. Both a compiler and an interpreter must act as if the translation phases in § 2.1.1.2 were followed. This is a requirement that an implementation act as if the entire program is translated before the program’s execution.

The ‘‘shall’’ in Footnote 63 in § 3.5.2.3 carries no special meaning: this footnote, like all other footnotes in the standard, is provided to emphasize the consequences of the rules in the standard. The footnote is not part of the standard.

The committee believes that a careful reading of the standard shows all of the places that the size of an object is needed, and that the translation phases prevent those requirements from being relaxed by an implementation.

Question 11: Clarification of incomplete `struct` declaration

Referring to § 3.5.2.3, page 63.

```
struct  t;
struct  t;  /*  Is this undefined?  */
```

People seem to think that the above is undefined.

The problem arises because no rules exist for compatibility of incomplete structures or unions.

Proposed resolution
Add either of the following paragraphs.

1) Two incomplete structures or unions are compatible if they are declared in the same scope, their tag names have the same spelling, and they both refer to structures or both to unions.

2) If two incomplete structures or unions occurring in the same scope have tags with the same spelling the behavior is undefined.

Arguments for/against
Opinions received fell into two categories:

1) Don’t care.

2) Ought to be allowed on the basis that the incomplete type may occur in multiple header files. Alternatively, machine generated code may produce such sequences.

Interpretation:

The proposed example is valid. Nothing in the standard prohibits it. Delete the ‘‘Proposed resolution’’.

The relevant citation is § 3.5.2.3 Semantics, paragraph 2:

> A declaration of the form
>
> *struct-or-union identifier* ;
>
> specifies a structure or union type and declares a tag, both visible only within the scope in which the declaration occurs. It specifies a new type distinct from any type with the same tag in an enclosing scope (if any).

Question 12: Ambiguous parsing of typedefs in prototypes

On page 68 in § 3.5.4.3, an ambiguity needs resolving in the parsing of the following:

a) `int x(T (U));`

b) `int x(T (U (int a, char b)));`

In a) `U` is the type of the parameter to a function returning type `T`. From § 3.5.4.3 page 69, line 2: ‘‘In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.’’

Thus in the case of b):

1) `U` could be a redundantly parenthesized name of a function which takes a *parameter-type-list* and returns type `T`, or

2) `U` could be the type returned by a function which takes a *parameter-type-list*, which in turn is the single parameter of a function returning type `T`.

Interpretation:

[This response duplicates that given to Request for Interpretation # 9, which asked the same question.]

It is widely recognized that the context-free syntax of C contains several instances where an identifier could be taken as either a typedef name or as an ordinary identifier. The following general principle has guided the description of such cases:

> Whenever a typedef name could be taken as such in a declaration, it is so taken.

The specific citations of this intent are in § 3.5.4.3 Semantics, paragraph 3:

> In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

and in § 3.5.6 Semantics, paragraph 1:

> If the identifier is redeclared in an inner scope or is declared as a member of a structure or union in the same or an inner scope, the type specifiers shall not be omitted in the inner declaration.

The committee acknowledges that the Request for Interpretation has identified another such context, to be disambiguated by the same general principle.

Question 13: Compatibility of functions with `register` on parameters

Reference § 3.5.4.3, page 68.

```
f1(int);
f1(register  int  a)  /*  Is this function compatible with the above?  */
{
}
```

§ 3.5.4.3 page 69, lines 5-7 were presumably intended to make sure that the `register` storage class got kept in the case of a definition so that the appropriate constraints applied, *i.e.*, it is not allowed to take its address, *etc.* But the further implication of the wording is that the occurrence of `register` lingers on for other uses — but there are no other uses.

Suggest a clarification on this point.

Proposed resolution
Add in § 3.5.4.3 on page 69 between line 7 and line 8:

> A storage-class specifier in a parameter type list of a function definition does not affect the compatibility of that definition with other function declarations not containing a storage-class specifier within their parameter type lists.

Interpretation:

The function is compatible. Storage class is not part of the type. Delete the ‘‘Proposed resolution’’.

The relevant citation, as given, is § 3.5.4.3, page 69, lines 5-7, but it does not imply any ‘‘other uses’’.

Question 14: `const void` type as a parameter

Refer to § 3.5.4.3, page 68, line 37. `f(const void)` should be explicitly undefined; also `f(register void)`, `f(volatile void)`, and combinations thereof.

Proposed resolution
Change § 3.5.4.3 on page 68, line 37 from

> The special case of `void` as the only ...

to

> The special case of the qualified or unqualified `void` type as the only...

Interpretation:

The net effect of the initial description is correct; if any storage-class specifiers or qualifiers modify `void`, the behavior is undefined. The committee would have no argument with saying ‘‘undefined behavior’’ explicitly.

The ‘‘Proposed resolution’’ contradicts this, and is incorrect.

The citation, as given, is § 3.5.4.3, page 68, line 37.

The key is that `void` in this context does not specify a type.

Question 15: Ordering of conversion of arrays to pointers

In § 3.5.4.3 on page 69 line 22 there is a sentence in parentheses. Does the sentence refer to the whole paragraph or just the preceding sentence?

```
int  f(int  a[4]);
int  f(int  a[5]);
int  f(int  *a);
```

- It refers to the whole paragraph.
  - — This makes all of the above three declarations compatible.
- It does not refer to the whole paragraph.
  - — This makes all three declarations incompatible.

Interpretation:

Regarding page 69 line 22: There are *two* sentences in parentheses. They apply to the entire paragraph. The declarations are all compatible.

Question 16: Pointer to multidimensional array

Given the declaration:

```
char  a[3][4],  (*p)[4]=a[1];
```

Does the behavior become undefined when:

a) `p` no longer points within the slice of the array, or

b) `p` no longer points within the object `a`?

This case should be explicitly stated.

Arguments for/against
The standard refers to a pointed-to object. There does not appear to be any concept of a slice of an array being an independent object.

Interpretation:

For an array of arrays, the permitted pointer arithmetic in Standard § 3.3.6 Semantics (page 48, lines 12-40) is to be understood by interpreting the use of the word ‘‘object’’ as denoting the specific object determined directly by the pointer’s type and value, *not* other objects related to that one by contiguity. For example, the following code has undefined behavior:

```
int  a[4][5];
a[1][7]  =  0;  /*  undefined  */
```

Some conforming implementations may choose to diagnose an ‘‘array bounds violation’’, while others may choose to interpret such attempted accesses successfully with the ‘‘obvious’’ extended semantics.

Question 17: Initialization of unions with unnamed members

§ 3.5.7 on page 72, line 38 says: ‘‘All unnamed structure or union members are ignored …’’ On page 73, lines 22-23 it says: ‘‘… for the first member of the union.’’ § 3.5.2.1, page 61, line 40 and Footnote 60 say that a field with no declarator is a member.

```
union  {
            int    :3;
            float  f;}  u  =  {3.4};
```

Should page 73 be changed to refer to the first named member or is the initialization of a union whose first member is unnamed illegal?

It has been suggested that the situation described above is implicitly undefined.

I think that it is a straightforward ambiguity that needs resolution one way or the other.

Proposed resolution
Make either of the following changes.

1) Change page 73, lines 22-23 from ‘‘… for the first member of the union’’ to ‘‘… for the first named member of the union’’.

2) Add on page 73 between lines 23-24

   If an object of union type with an unnamed first member is initialized the behavior is undefined.

Interpretation:

The statement in § 3.5.7 on page 72, line 38 of the standard takes precedence. That is, unnamed members do not correspond to explicit initializers.

Question 18: Compatibility of functions with `void` and no prototype

```
f2(void);
f2();  /*  Is this function compatible with the one above?  */
```

Now § 3.5.4.3 page 69, line 1 says that the first declaration of `f1` specifies that the function has no parameters.

No rules are given in the subsequent paragraphs to say that a function declaration with a parameter type list, with no parameters, is compatible with a function declaration with an empty parameter list.

If we treat the `void` as a single parameter then page 69 lines 15-18 would make the above two functions incompatible. `void` is not compatible with any default promotions. § 3.5.4.3 page 69 lines 18-22 cover the case for declaration and definition.

Thus I think that in the above example the behavior is implicitly undefined.

Proposed resolution
Add on page 69 between lines 25-26

If one type has a parameter type list containing `void` then the other type shall have an empty parameter list.

Interpretation:

The standard on page 68, line 37 and page 69, line 1 states ‘‘The special case of `void` as the only item in the list specifies that the function has no parameters.’’ Therefore, in the case of `f2(void);` there are *no* parameters just as there are none for `f2();`. Since both functions have the same return type, these declarations *are* compatible.

Question 19: Order of evaluation of macros

Refer to § 3.8.3, page 90. In:

```
#define  f(a)  a*g
#define  g(a)  f(a)
f(2)(9)
```

it should be defined whether this results in:

a) `2*f(9)`, or

b) `2*9*g`

X3J11 previously said, ‘‘The behavior in this case could have been specified, but the committee has decided more than once not to do so. [They] do not wish to promote this sort of macro replacement usage.’’

I interpret this as saying, in other words, ‘‘if we don’t define the behavior nobody will use it.’’ Does anybody think this position is unusual?

People seem to agree that the behavior is ambiguous in this case. Should we specify this case as undefined behavior?

Proposed resolution
Add one of the following paragraphs to page 90.

1) If a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token it is undefined whether this macro name may be subsequently replaced.

2) If a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token then this macro name may be replaced.

3) If a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token then this macro name may not be replaced.

Interpretation:

The committee addressed this issue explicitly in previous deliberations and decided to say nothing about the situation, understanding that behavior in such cases would be undefined.

Question 20: Scope of macro parameters

Refer to § 3.8.3 on page 90, line 16; the scope of macro parameters should be defined in the section on scope.

The idea is to enable all references to the scope of names to be under one heading. This is not really a significant issue.

Proposed resolution
Move § 3.8.3 page 90, line 18 to § 3.1.2.1 after page 21, line 42.

Interpretation:

In the standard in § 3.1.2 on page 21, line 5 it states ‘‘Macro names and macro parameters are not considered further here. [(§ 3.1.2)]’’ This approach was intentionally adopted to avoid explicitly having to mention exceptions of using identifiers, for example in the sections on scope, linkage, name spaces, and storage durations, none of which applies to macros. The proposed change does *not* clarify the standard and may even obscure it.

Question 21: Self references in translation phase 4

The following queries arise because of the imprecise way in which phase 4 interacts with itself. While processing a token within phase 4 it is sometime necessary to get the following tokens from the input, *i.e.* reading the arguments to a function-like macro. But when getting these tokens it is not clear how many phases operate on them:

1) Do the following tokens only get processed by phases 1-3?

2) Do the following tokens get processed by phases 1-4?

When an identifier declared as a function-like macro is encountered, how hard should an implementation try to locate the opening/closing parentheses?

In:

```
#define  lparen  (
#define  f_m(a)  a
f_m  lparen  "abc"  )
```

should the object-like macro be expanded while searching for an opening parenthesis? Or does the lack of a readily available left parenthesis indicate that the macro should not be expanded?

§ 3.8.3, on page 90, lines 34-35, says ‘‘… followed by a `(` as the next preprocessing token …’’. This sentence does not help because in translation phase 4 all tokens are preprocessing tokens. They don’t get converted to ‘‘real’’ tokens until phase 7. Thus it cannot be argued that `lparen` is not correct in this situation, because its result is a preprocessing token.

In:

```
#define  i(x)  3
#define  a  i(yz
#define  b  )
a  b  ) /*  goes to 3) or 3  */
```

does `b` get expanded to complete the call `i(yz`, or does the parenthesis to its right get used?

Proposed resolution
Add either of the following paragraphs to page 90:

1) While searching for the opening parenthesis in a function-like macro replacement or the matching closing parenthesis the preprocessing tokens examined are only processed by translation phases 1 through 3.

2) Whether the preprocessing tokens examined while searching for the opening parenthesis in a function-like macro replacement or its matching closing parenthesis are themselves subject to macro replacement is undefined.

Comments received suggested that no other macros were initially expanded while processing a function like macro.

Interpretation:

Concerning the first example:

```
#define  lparen  (
#define  f_m(a)  a
f_m  lparen  "abc"  )
```

According to § 2.1.1.2 (‘‘Translation Phases’’) page 6, lines 25-39, the translation phases 1-3 do not cause macros to be expanded. Phase 4 does expand. To apply § 3.8.3 (‘‘Macro Replacement’’) page 90, lines 34-35 to the example: Since `lparen` is not `(` in ‘‘`f_m lparen "abc" )`’’, this construct is not recognized as a function-like macro invocation. Therefore the example expands to

```
f_m("abc")
```

The same principle applies to the second example:

```
#define  i(x)  3
#define  a  i(yz
#define  b  )
a  b  )              /*  expands via the following stages:  */
i(yz  b  )         /*  ) delimits the argument list before b is expanded  */
i([yz  )  ])
3
```

This is how we interpret § 3.8.3, page 90, lines 36-38: The sequence of preprocessing tokens is terminated by the right-parenthesis preprocessing token.

Question 22: Gluing during rescan

Reference: page 91. Does the rescan of a macro invocation also perform gluing?

```
#define  hash_hash  #  ##  #
#define  mkstr(a)  #  a
#define  in_between(a)  mkstr(a)
#define  join(c,  d)  in_between(c  hash_hash  d)
char  p[2]  =  join(x,  y);
```

Is the above legal? Does `join` expand to `"xy"` or `"x ## y"`?

It all depends on the wording in § 3.8.3.3 on page 91, lines 39-40. Does the wording ‘‘… before the replacement list is reexamined …’’ mean before being reexamined for the first time only, or before being reexamined on every rescan?

This rather perverse macro expansion is only made possible because the constraints on the use of # refer to function-like macros only. If this constraint were extended to cover object-like macros the whole question goes away.

Dave Prosser says that the intent was to produce `"x ## y"`. My reading is that the result should be `"xy"`. I cannot see any rule that says a created ## should not be processed appropriately. The standard does say on page 91 line 40 ‘‘… each instance of a ## …’’.

The reason I ask if the above is legal is that the order of evaluation of # and ## is not defined. Thus if # is performed first the result is very different than if ## goes first.

Proposed resolution
Make one of the following changes.

1) Add to § 3.8.3.2 Constraints on page 91 between lines 18-19

   The # preprocessing token may not occur in the replacement list of an object-like macro.

2) Change page 91, line 40 from ‘‘… for more macro names to replace’’ to ‘‘… for more macro names to replace the first time’’

3) Change page 91, line 39 from ‘‘… before the replacement list …’’ to ‘‘… every time before the replacement list …’’

Interpretation:

The Request for Interpretation poses this puzzle:

```
#define  hash_hash  #  ##  #
#define  mkstr(a)  #  a
#define  in_between(a)  mkstr(a)
#define  join(c,  d)  in_between(c  hash_hash  d)
join(x,  y)
```

Does this produce `"xy"` or `"x ## y"`?

Answer: The expansion produces, at various stages:

```
join(x,  y)
in_between(x  hash_hash  y)
in_between(x  ##  y)
mkstr(x  ##  y)
"x  ##  y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp-signs, but this new token is not the catenation operator.

Question 23: How long does blue paint persist?

Consider the following code:

```
#define  a(x)  b
#define  b(x)  x
    a(a)(a)(a)
```

The macro replacement for `a(a)` results in `b`.

First replacement buffer: `b` Remaining tokens: `(a)(a)`

Inside the first replacement buffer, no further nested replacements will recognize the macro name ‘‘`a`’’. The name ‘‘`a`’’ is painted blue.

The first replacement buffer is rescanned not by itself, but along with the rest of the source program’s tokens. ‘‘`b(a)`’’ also causes macro replacement and becomes ‘‘`a`’’.

Second replacement buffer: `a` Remaining tokens: `(a)`

The second replacement buffer is rescanned not by itself, but along with the rest of the source program’s tokens.

The ‘‘`a`’’ in the second replacement buffer did not come from the first replacement buffer. It came from three of the remaining tokens which were in the source file following the first replacement buffer. Is this ‘‘`a`’’ part of a nested replacement? Is it still painted blue?

Note that there are many ‘‘paths’’ that can be taken for a possible macro name to travel from a preprocessing token (outside the replacement buffer) to one that is inside the replacement buffer. When do they stop getting painted blue? If either too early or too late, they cause very surprising results.

Given the amount of discussion involving macro expansion that uses the concept of ‘‘blue paint’’, why doesn’t the standard tell the reader about this idea?

Everybody seems to agree that the above is undefined. Does anybody have a set of words to make this and other cases explicitly undefined?

Interpretation:

The reference is to § 3.8.3.4, page 92.

```
#define  a(x)  b
#define  b(x)  x
a(a)(a)(a)      /*  expands as follows:  */

b(a)(a)
a´(a)    or       a(a)
a(a)              b
```

[a´ indicates the symbol a marked for non-replacement]

The committee addressed this issue explicitly in previous deliberations and decided to say nothing about the situation, understanding that behavior in such cases would be undefined.

The result, as with other examples, is intentionally left undefined.

Question 24: Improve English

Just a tidy up.

Proposed resolution
Change § 4.1.2, page 97, line 34 from ‘‘if the identifier’’ to ‘‘if an identifier’’.

Interpretation:

The standard is clear enough as is.

Question 25: ‘‘Must’’ in footnotes

This change is not essential since footnotes have no status, other than creating explicitly undefined behavior. But this change would cut down the number of occurrences of ‘‘shall’’ synonyms used where ‘‘shall’’ itself could have be used.

Proposed resolution
In Footnote 91 on page 98 (§ 4.1.2.1) change ‘‘must not’’ to ‘‘shall not’’.

Interpretation:

The standard is clear enough as is.

Question 26: Implicit initialization of unions with unnamed members

Proposed resolution
On page 73, lines 1-2, change both occurrences of ‘‘… every member’’ to ‘‘… every named member’’.

Interpretation:

Are unnamed union members required to be initialized?

No.

§ 3.5.7 page 72, line 38 says ‘‘All unnamed structure or union members are ignored during initialization.’’ Therefore it is not necessary to modify the wording of the standard because a careful reading yields the correct answer. We recommend against any changes to the base document for clarification. Clarifying notes may be appropriate for addenda that are not part of the base document.

Question 27: `g` conversions

§ 4.9.6.1 says on page 133, lines 42-43: ‘‘For `g` and `G` conversions, trailing zeros will *not* be removed …’’, whereas on page 134, lines 37-39 it says: ‘‘Trailing zeros are removed …’’

Proposed resolution
Make either of the following changes.

1) On page 134 line 38 change ‘‘… portion of the result’’ to ‘‘… portion of the result unless the # flag is specified’’
2) On page 133 lines 42-43, remove ‘‘For `g` and `G` conversions, trailing zeros will *not* be removed from the result.’’

Arguments for/against
It has been suggested that the italics on page 133, lines 42-43 gives this rule precedence. I don’t mind which rule wins as long as the text says so. Do we add text to describe the italics rule or change the conflicting lines?

Interpretation:

In the collision between the description of the # flag and the `g` and `G` conversion specifiers to `fprintf`, which takes precedence?

The # flag takes precedence.

§ 4.9.6.1 page 133, line 1 says ‘‘• Zero or more *flags* (in any order) … modify the meaning of the conversion specification.’’

There is no such thing as an ‘‘italics rule’’.

Question 28: Ordering of conditions on return

Proposed resolution
In § 4.9.9.1 on page 146, lines 25-26 change ‘‘… returns nonzero and stores …’’ to ‘‘… stores an implementation-defined positive value in `errno` and returns nonzero’’.

In § 4.9.9.3 on page 147, lines 16-17 change ‘‘… returns nonzero and stores …’’ to ‘‘… stores an implementation-defined positive value in `errno` and returns nonzero’’.

In § 4.9.9.4 on page 147, lines 31-32 change ‘‘… returns -1L and stores an …’’ to ‘‘… stores an implementation-defined positive value in `errno` and returns -1L’’.

Interpretation:

The question is, ‘‘Strange order of operations — shouldn’t the wording be reversed?’’

No.

In § 4.9.9.1, § 4.9.9.3, and § 4.9.9.4, the words ‘‘returns … and stores an implementation-defined positive value in `errno`’’ do not imply any temporal ordering. There are implementations that may perform these operations in either order and they still meet the standard.

Question 29: Conversion failure and longest matches

Consider `1.2e+4` with field width of 5. Is it input item `1.2e+` that gives a conversion failure? What is the ordering between building input items and converting them? Do they run in parallel, or sequential?

Refer to § 4.9.6.2 (‘‘The `fscanf` Function’’) page 136 lines 31-33 concerning the longest matching sequence, and § 4.9.6.2, page 138 lines 15-16 concerning a conflicting input character.

For `1.2e-x`, is `1.2` or `1.2e-` read?

The above questions all come about because of page 138 line 11: ‘‘If conversion terminates …’’. In this context the use of the word ‘‘conversion’’ could be referring to the process of turning a sequence of characters into numeric form. I believe what was intended was ‘‘If a conversion specifier terminates …’’.

Interpretation:

The relevant citations are § 4.9.6.2, page 138, lines 15-16

> If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream.

and § 4.9.6.2, page 136, lines 31-33

> An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence.

and § 4.9.6.2, page 136, lines 38-40

> If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure.

The ‘‘conversion’’ in the first quoted passage is the process of both forming an input item and converting it as specified by the conversion specifier.

About your example: If the characters available for input are ‘‘`1.2e+4`’’ and input is performed using a ‘‘`%5e`’’, then the input item is ‘‘`1.2e+`’’ as defined by the second passage quoted above. That input item is not a matching sequence, but only an initial subsequence that fails to be a matching sequence in its own right. Under the rules of the third quoted passage, this is a matching failure.

Note that in this case, no characters were pushed back on to the input stream. There was no ‘‘conflicting input character’’ that terminated the field, and so the first quoted passage does not apply.

Question 30: Successful call to `ftell` or `fgetpos`

Proposed resolution
In § 4.9.9.2 on page 146, lines 39-40 change ‘‘… a value returned by an earlier call to the `ftell` function …’’ to ‘‘… a value returned by an earlier successful call …’’.

In § 4.9.9.3 on page 147, lines 10-11 change ‘‘… a value obtained from an earlier call to the `fgetpos` function …’’ to ‘‘… a value obtained from an earlier successful call …’’.

Interpretation:

The proposed addendum wording is reasonable; this is the intent of the standard.

Question 31: Size in bytes

References to the size of an object in other parts of the standard specify that size is measured in bytes. The following lines do not follow this convention.

Proposed resolution
In § 4.10.3.1 on page 155, lines 26-27 and in § 4.10.3.3 on page 156, line 8 change ‘‘whose size is …’’ to ‘‘whose size (in bytes) is …’’.

Interpretation:

There are numerous places in the standard where ‘‘size in bytes’’ is used, and numerous places where ‘‘size’’ alone is used. The committee does not feel that any of these places need fixing — the meaning is everywhere clear, especially since for `sizeof` in § 3.3.3.4 size is specifically mentioned in terms of bytes. With regard to the proposed addendum, for the addendum to clean up only a few of the places, as proposed, we think is not reasonable; ‘‘addendumize’’ them all or leave them all alone.

Question 32: `char` parameters to `strcmp` and `strncmp`

Refer to § 4.11.4, page 165. If `char` is signed then `char *` cannot be interpreted as pointing to `unsigned char`. The required cast may give undefined results. This applies to `strcmp` and `strncmp`.

Proposed resolution
Add to § 4.11.4, page 165:

> In an implementation that defines `char` to be signed the result of this comparison will be undefined if the value of any of the characters cannot be represented in an `unsigned char`.

Interpretation:

`strcmp` can compare two `char` strings, even though the representation of `char` may be signed, because § 4.11.4, page 165 line 7 says that the interpretation of bytes is done as if each byte were accessed as an `unsigned char`. We believe the standard is clear and believe there is no need for an addendum item.

Question 33: Different length strings

Refer to § 4.11.4, page 165 lines 5-7. What about strings of different length?

Perhaps the fact that the terminating null character takes part in the comparison ought to be mentioned.

Interpretation:

§ 4.1.1, on page 97, lines 5-6 says that a string includes the terminating null character. Therefore this character takes part in the comparison. The standard is clear and there is no need for an addendum item.

Question 34: Calls to `strtok`

In § 4.11.5.8 on page 168 line 36, ‘‘… first call …’’ should read ‘‘… all calls …’’.

I think that the current wording causes confusion. The first call is the one that takes a non-`NULL` ‘‘`s1`’’ parameter. However, the discussion from line 36 onwards is describing the behavior for all calls.

Interpretation:

The committee felt that the suggested wording for the `strtok` function description is not an improvement. The existing wording is clear as written.

Question 35: When is a physical source line created?

Is the output or input to translation phase 1 a physical source line?

Interpretation:

The use of the term ‘‘physical source line’’ occurs only in the description of the phases of translation (§ 2.1.1.2) and the question of whether the input or output of phase 1 consists of physical source lines does not matter.

Question 36: Qualifiers on function return type

Refer to § 3.6.6.4, page 81, line 24: ‘‘… whose return type is `void`.’’

The behavior of a type qualifier on a function return is explicitly undefined, according to § 3.5.3, page 65 lines 24-25.

This creates a loophole.

An implementation that supports type qualifiers on function return types is not required to flag the constraint given on page 81.

Proposed resolution
In § 3.6.6.4 on page 81 lines 24-25 change ‘‘… whose return type is `void`’’ to ‘‘… whose unqualified return type is `void`’’.

Interpretation:

The question mentions text on page 65, lines 24-25 as pertaining to function return types. In fact, that portion of the standard refers to function types, not return types, and so is not relevant to the question.

A function returning a qualified version of type `void` was intentionally left out of the constraint on page 81, line 24. This means that the behavior of such functions is implicitly undefined.

Question 37: Function result type

Reference: § 3.3.2.2, page 41 line 35.

The result type of a function call is not defined.

Proposed resolution
It should be made explicitly undefined.

Interpretation:

[No interpretation was available at the time of publication.]

Question 38: What is an iteration control structure or selection control structure?

An ‘‘iteration control structure’’, a term used in § 2.2.4.1 (‘‘Translation Limits’’) on page 14 line 1, is not defined by the standard.

Is it:

1) A `for` loop header excluding its body, *e.g.* `for (;;)`, or
2) A `for` loop header plus its body, *e.g.* `for (;;) {}`?

Does it make a difference if the compound statement is a simple statement without the braces?

Interpretation:

The committee’s opinion was that the term ‘‘iteration control structure’’ was the same as ‘‘iteration statement’’, which is defined by the standard. As a result, this means that the statement which is the loop body is considered part of the ‘‘iteration control structure’’. Similarly a ‘‘selection control structure’’ is the same as a selection statement.

In discussing nesting levels, the consensus of the committee was that the fragment

```
for  (...)
     for  (...)
```

contained 2 nesting levels while the fragment

```
for  (...)  {
     for  (...)
```

contained 3 nesting levels (*i.e.*, the introduction of the `{` implied another nesting level).

Question 39: Header name tokenization

There is an inconsistency between § 3.1.7 page 34 line 8 and the description of the creation of header name preprocessing tokens.

The ‘‘shall’’ on page 33 line 33 does not limit the creation of header name preprocessing tokens to within `#include` directives. It simply states that they would cause a constraint error in this context.

§ 3.1.7, page 34 line 8 should read {`0x3`}{`<1/a.h>`}{`1e2`}, or extra text needs to be added to § 3.1.7.

I have not met anybody who expects `if (a<b || c>d)` to parse as:

{`if`} {`(`} {`a`} {`<b || c>`} {`d`} {`)`}

Interpretation:

Footnote 5 in § 2.1.1.2 on page 6 indicates that tokenization is context-dependent, and so does the example. The constraint on page 33 line 33 means that `<...>` is tokenized specially only on `#include` lines. In other words, this constraint is intended to be applied to the implementation, not the translation unit.

Request for Interpretation # 18 (X3J11 Doc. No. 90-066)

Question 1:

It is unclear how the `fscanf` function shall behave when executing directives that include ‘‘ordinary multibyte characters’’, especially in the case of shift-encoded ordinary multibyte characters.

The following statements are described in § 4.9.6.2 (‘‘The `fscanf` Function’’) of the current standard.

> A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

Assume a typical shift-encoded directive: Ä in 7-bit representation. And, consider two different encoding systems, Latin Alphabet No.1 — 8859/1 and German Standard DIN 66 003. The codes are, for example,

> Ä in 8859/1: SO 4/4 SI
> Ä in DIN 66 003: ESC 2/8 4/11 5/11 ESC 2/8 4/2
>
> where SO is a Shift-Out code (0/15 = 0x0F) and SI corresponds to a Shift-In code (0/14). ‘‘ESC 2/8 4/11’’ is an escape sequence for the German Standard DIN 66 003, and ‘‘ESC 2/8 4/2’’ is for ISO 646 USA Version (ASCII).

Assuming that a subject sequence includes Ä, Ö, and Ü with the following 7-bit representations,

> In 8859/1: SO 4/4 5/6 5/12 SI
> In DIN 66 003: ESC 2/8 4/11 5/11 5/12 5/13 ESC 2/8 4/2

does the ‘‘Ä’’ directive in the `fscanf` format string match the beginning part of the ‘‘ÄÖÜ’’ sequence?

At what position of the target sequence shall the ‘‘Ä’’ directive fail?

One interpretation of this is that because the current standard defined the behavior of the directive in the `fscanf` format based on the word ‘‘character’’ (byte), not using the term ‘‘multibyte character’’, the comparison shall be done on a byte-by-byte basis. One may conclude that the ‘‘Ä’’ directive never matches the ‘‘ÄÖÜ’’ sequence in this case.

Another interpretation may lead to an opposite conclusion, saying that the current standard’s statements quoted above do not necessarily mean that such comparison shall be done on a byte-by-byte basis. Instead, it is read that the matching shall be done on a ‘‘multibyte character by multibyte character basis’’ or rather ‘‘wide character by wide character basis’’. Especially, a ‘‘ghost’’ sequence like ‘‘ESC …’’ and SI/SO characters should not be regarded as independent ordinary multibyte characters in this case.

Which is a correct interpretation of the current standard?

These different interpretations are caused by the ambiguity of the descriptions in the current standard. Also, it should be pointed out that the major problem here is usage of the word ‘‘character’’. The generic word ‘‘character’’ and the specific word ‘‘character(=byte)’’ should be properly discriminated in the standard.

Interpretation:

§ 4.9.6.2 says, ‘‘A directive that is an ordinary multibyte character is executed by reading the next *characters* …’’ [emphasis added]. Consistently throughout the standard, plain ‘‘characters’’ refers to one-byte characters. (See § 1.6 for the definition of ‘‘character’’.)

Request for Interpretation # 19 (X3J11 Doc. No. 91-014)

Question 1:

Background:

§ 4.3.1.5 states that ‘‘the `isgraph` function tests for any printing character except space’’. § 4.3.1.7 states that ‘‘the `isprint` function tests for any printing character including space’’.

The third paragraph of § 4.3 defines the term *printing character* as ‘‘a member of an implementation-defined set of characters, each of which occupies one printing position on a display device’’.

§ 2.2.1 defines the source and execution character sets and provides a list of characters which must be contained in both sets.

Question for interpretation:

Are the `isprint` and `isgraph` functions required to return a non-zero value for all of the characters defined in § 2.2.1?

A scenario for use of `isprint`/`isgraph` that depends on the interpretation is:

A developer may wish to use these functions to determine whether a particular character can be displayed as itself (*e.g.*, whether a square bracket is actually displayed as a square bracket). This could be useful for formatting output in a device-independent manner, since the application could substitute some other character for ones that do not print ‘‘correctly’’.

If `isprint` and `isgraph` are required to return non-zero for all characters in § 2.2.1, developers cannot use them for this purpose.

This problem has occurred in a real implementation. The most commonly used terminals and printers for IBM System/370 computers do not support all of the characters listed in § 2.2.1. For example, most IBM printers and terminals do not print the square brackets.

The SAS/C implementation of `isprint` and `isgraph` assumes that § 4.3 controls the behavior of these functions, and returns non-zero only for those characters that print ‘‘correctly’’. The Plum Hall test suite, however, assumes that `isprint` and `isgraph` return non-zero for all characters listed in § 2.2.1.

Interpretation:

§ 4.3 page 103, line 8 says that *printing character* is implementation-defined. In particular, the value (zero or non-zero) of `isprint('[')` is implementation-defined, *even in the* `"C"` *locale*.

Request for Interpretation # 20 (X3J11 Doc. No. 91-006)

Question 1:

Is a compiler which allows the Relaxed Ref/Def linkage model to be considered a conforming compiler? That is, can a compiler that compiles the following code with no errors or warnings

filea.c:

```
#include <stdio.h>
void foo(void);
int age;
void main()
{
     age = 24;
     printf("my age is %d.\n", age);
     foo();
     printf("my age is %d.\n", age);
     return 0;
}
```

fileb.c:

```
#include <stdio.h>
int age;
void foo()
{
     age = 25;
     printf("your age is %d.\n", age);
}
```

and which produces the following output

```
my age is 24
your age is 25
my age is 25
```

be called a standard-compliant compiler?

Interpretation:

Yes, a compiler that allows the Relaxed Ref/Def model can be standard conforming. See § 3.7, page 82, lines 23-25: the code is conforming but not strictly conforming. The behavior is undefined.

Request for Interpretation # 21 (X3J11 Doc. No. 91-001)

Question 1:

What is the result of: `printf( "%#.4o", 345 );`? Is it ‘‘`0531`’’ or is it ‘‘`00531`’’?

§ 4.9.6.1, on page 133, lines 37-38 says: ‘‘For `o` conversion, it increases the precision to force the first digit of the result to be a zero.’’

Is this a conditional or an unconditional increase in the precision if the most significant digit is not already a `0`? Which is the correct interpretation?

Interpretation:

In the format `%#.4o`, the precision is increased *only if necessary* [to obtain `0` for the first digit].

Request for Interpretation # 22 (X3J11 Doc. No. 91-002)

Question 1:

What is the result of: `strtod( "100ergs", &ptr);`? Is it 100.0 or is it 0.0?

§ 4.10.1.4 (‘‘The `strtod` Function’’) on page 151, lines 36-38 says: ‘‘The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form.’’ In this case, the longest initial subsequence of the expected form is ‘‘`100`’’, so 100.0 should be the return value. Also, since the entire string is in memory, `strtod` can scan it as many times as need be to find the longest valid initial subsequence.

§ 4.9.6.2 (‘‘The `fscanf` Function’’) on page 137, lines 17-18 says: ‘‘`e`,`f`,`g` Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function.’’ Later, page 139, line 6, line 16, and line 25 show that ‘‘`100ergs`’’ fails to match ‘‘`%f`’’. Those two show that ‘‘`100ergs`’’ is invalid to `fscanf` and therefore, invalid to `strtod`. Then, § 4.10.1.4 page 152, lines 11-12’s ‘‘If no conversion could be performed, zero is returned’’ indicates for an error input, 0.0 should be returned. The reason this is invalid is spelled out in the rationale document, § 4.9.6.2 (The `fscanf` function), page 95: ‘‘One-character pushback is sufficient for the implementation of `fscanf`. Given the invalid field ‘‘`-.x`’’, the characters ‘‘`-.`’’ are not pushed back.’’ And later, ‘‘The conversions performed by `fscanf` are compatible with those performed by `strtod` and `strtol`.’’

So, do `strtod` and `fscanf` act alike and both accept and fail on the same inputs, by the one-character pushback scanning strategy, or do they use different scanning strategies and `strtod` accept more than `fscanf`?

Interpretation:

`strtod` and `fscanf` are different — `fscanf` expects a sequence that matches what works for `strtod`. `fscanf` can easily get into corners that it can’t back out of, as it is constrained not to ‘‘leave unread’’ more than one byte. On the other hand, `strtod` must do a reasonable job with any input string.

Thus, `fscanf(fp, "%f", &f)` when handed ‘‘`100ergs`’’ on its input stream will fail to convert, lose ‘‘`100e`’’, and leave ‘‘`rgs`’’ unread (§ 4.9.6.2). In contrast, `strtod("100ergs", &ptr)` returns 100.0 and sets the pointer to point to the ‘‘`e`’’ character (§ 4.10.1.4).

Request for Interpretation # 23 (X3J11 Doc. No. 91-003)

Question 1:

Assuming that 99999 is larger than `DBL_MAX_10_EXP`, what is the result of:

```
strtod("0.0e99999",  &ptr);
```

— is it 0.0, `HUGE_VAL`, or undefined?

§ 3.1.3.1 (‘‘Floating Constants’’) on page 27, lines 30-32 says: ‘‘The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of 10 by which the significand part is to be scaled.’’ In this case `0.0e99999` means 0.0 times 10 to the power 99999, or 0.0×10^{99999}, which has a scaled value of 0.0; therefore, return 0.0.

§ 4.10.1.4 (‘‘The `strtod` Function’’) on page 152, lines 12-14 says: ‘‘If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`.’’ Since the exponent (99999 in this case) is larger than `DBL_MAX_10_EXP`, the value is outside the range of representable values (overflow). Therefore, return `HUGE_VAL`.

§ 2.2.4.2.2 (‘‘Characteristics of Floating Types `<float.h>`’’) page 15-16 describe the model that defines the floating-point types. The number `0.0e99999`, as written, is not part of that model (it cannot be represented since the exponent is larger than e_{max}). From § 3.2.1.4 (‘‘Floating Types’’) page 36, lines 11-13, ‘‘… if the value being converted is outside the range of values that can be represented, the behavior is undefined.’’ Therefore, since this number, as written, has no representation, the behavior is undefined.

Interpretation:

According to our interpretation for Request for Interpretation # 25 (X3J11 Document Number 91-005), the result of `strtod("0.0e99999", &ptr)` is exactly representable, *i.e.* lies within the range of representable values. Therefore, by § 4.10.1.4 Returns, the value zero shall be returned in this case, and `errno` shall not be set. (This means that implementations have to test for the special case of zero when creating floating-point representations from characters.)

Note also that `strtod("0.0e-99999", &ptr)` is not a case of underflow, so `errno` shall not be set to `ERANGE` in this case either.

Request for Interpretation # 24 (X3J11 Doc. No. 91-004)

Question 1:

In § 4.10.1.4 (‘‘The `strtod` Function’’) page 152, line 5: What does ‘‘`"C"` locale’’ mean?

a) `setlocale(LC_ALL,NULL) == "C"`

b) `setlocale(LC_NUMERIC,NULL) == "C"`

c) a) && b)

d) a) || b)

e) something else.

What does ‘‘other than the `"C"` locale’’ mean?

a) `setlocale(LC_ALL,NULL) != "C"`

b) `setlocale(LC_NUMERIC,NULL) != "C"`

c) a) && b)

d) a) || b)

e) something else.

§ 4.4.1 (‘‘Locale Control’’) page 108 may help answer the questions.

Interpretation:

§ 4.4.1 page 108, lines 11-17 describe what is affected by each locale portion. Is it the `LC_NUMERIC` locale category which affects the implementation-defined behavior of `strtod` *etc.*? Answer: Yes.

How can one guarantee that `strtod` functions are in the `"C"` locale? Answer: Execute `setlocale(LC_NUMERIC, "C")` or execute `setlocale(LC_ALL, "C")`.

What is meant by ‘‘other than the `"C"` locale’’? *I.e.*, how can one ensure that `strtod` is not in the `"C"` locale? Answer: Successfully execute `setlocale(LC_NUMERIC, str)` or `setlocale(LC_ALL, str)` to some implementation-defined string `str` which specifies a locale that is different from the `"C"` locale. No universally portable method can be provided, because the functionality is implementation-defined.

Request for Interpretation # 25 (X3J11 Doc. No. 91-005)

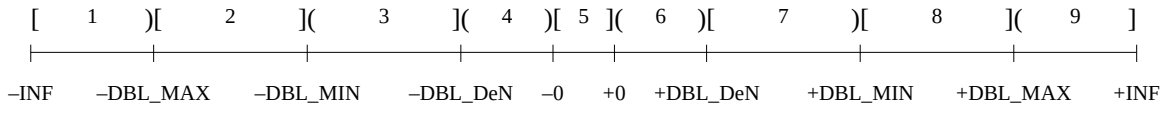
Question 1:

What is meant by ‘‘representable floating-point value’’? Assume double precision, unless stated otherwise.

First, some definitions based partially upon the floating-point model on page 15 of the C standard:

1) +Normal Numbers: `DBL_MIN` to `DBL_MAX`, inclusive; normalized (first significand digit is non-zero), sign is +1.
2) –Normal Numbers: `-DBL_MAX` to `-DBL_MIN`, inclusive; normalized.
3) +Zero: All digits zero, sign is +1; (true zero).
4) –Zero: All digits zero, sign is –1.
5) Zero: Union of +zero and –zero.
6) +Denormals: Exponent is ‘‘minimum’’ (biased exponent is zero); first significand digit is zero; sign is +1. These are in range `+DBL_DeN` (inclusive) to `+DBL_MIN` (exclusive). (Let `DBL_DeN` be the symbol for the minimum positive denormal, so we can talk about it by name.)
7) –Denormals: same as +denormals, except sign, and range is `-DBL_MIN` (exclusive) to `-DBL_DeN` (inclusive).
8) +Unnormals: Biased exponent is non-zero; first significand digit is zero; sign is +1. These overlap the range of +normals and +denormals.
9) –Unnormals: Same as +unnormals, except sign; range is over –normals and –denormals.
10) +infinity: >From IEEE-754.
11) –infinity: >From IEEE-754.
12) Quiet NaN (Not a Number); sign does not matter; from IEEE-754.
13) Signaling NaN; sign does not matter; from IEEE-754.
14) NaN: Union of Quiet NaN and Signaling NaN.
15) Others: Reserved (VAX?) and Indefinite (CDC/Cray?) act like NaN.

On the real number line, these symbols order as:



Non-real numbers are: SNaN, QNaN, and NaN; call this region 10.

Regions 1 and 9 are overflow, 2 and 8 are normal numbers, 3 and 7 are denormal numbers (pseudo underflow), 4 and 6 are true underflow, and 5 is zero.

So, the question is: What does ‘‘representable (double-precision) floating-point value’’ mean:

a) Regions 2, 5 and 8 (+/– normals and zero)
b) Regions 2, 3, 5, 7, and 8 (+/– normals, denormals, and zero)

c) Regions 2 through 8 [`-DBL_MAX` ... `+DBL_MAX`]

d) Regions 1 through 9 [–INF … +INF]

e) Regions 1 through 10 (reals and non-reals)

f) What the hardware can represent

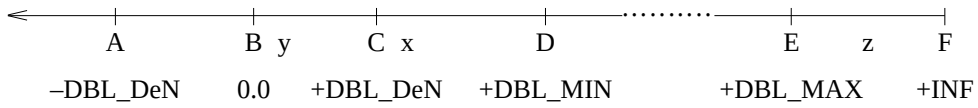
g) Something else? What?

Some things to consider in your answer follow. The questions that follow are rhetorical and do not need answers.

§ 2.2.4.2.2 (‘‘Characteristics of Floating Types `<float.h>`’’) page 15, lines 32-35: ‘‘The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation’s floating-point arithmetic.’’ Same section, page 16 line 6: ‘‘A normalized floating-point number *x* … is defined by the following model: …’’.

> That model is just normalized numbers and zero (appears to include signed zeros). It excludes denormal and unnormal numbers, infinities, and NaNs. Are signed zeros required, or just allowed?

§ 3.1.3.1 (‘‘Floating Constants’’) page 27, lines 32-35: ‘‘If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest value, chosen in an implementation-defined manner.’’



The representable numbers are A, B, C, D, E, and F. The number x can be converted to B, C, or D! But what if B is zero, C is `DBL_DeN` (denormal), and D is `DBL_MIN` (normalized). Is x representable? It is not in the range `DBL_MIN..DBL_MAX` and its inverse causes overflow; so those say not valid. On the other hand, it is in the range `DBL_DeN..DBL_MAX` and it does not cause underflow; so those say it is valid.

What if B is zero, A is `-DBL_DeN` (denormal), and C is `+DBL_DeN` (denormal); is y representable? If so, its nearest value is zero, and the immediately adjacent values include a positive and a negative number. So a user-written positive number is allowed to end up with a negative value!

What if E is `DBL_MAX` and F is infinity (on a machine that uses infinities, IEEE-754)? Does z have a representation? If z came from 1.0/x, then z caused overflow which says invalid. But on IEEE-754 machines, it would either be `DBL_MAX` or infinity depending upon the rounding control, so it has a representation and is valid.

What is ‘‘nearest’’? In linear or logarithmic sense? If the number is between 0 and `DBL_DeN`, *e.g.*, 10^{-99999}, it is linear-nearest to zero, but log-nearest to `DBL_DeN`. If the number is between `DBL_MAX` and INF, *e.g.*, 10^{+99999}, it is linear- and log-nearest to `DBL_MAX`. Or is everything bigger than `DBL_MAX` nearest to INF?

§ 3.2.1.3 (‘‘Floating and Integral’’) page 36, Footnote 29: ‘‘Thus, the range of portable floating values is (–1,U*type*_MAX+1).’’

§ 3.2.1.4 (‘‘Floating Types’’) page 36, lines 11-15: ‘‘When a `double` is demoted to `float` or a `long double` to `double` or `float`, if the value being converted is outside the range of values that can be represented, the behavior is undefined. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen in an implementation-defined manner.’’

§ 3.3 (‘‘Expressions’’) page 39, lines 15-17: ‘‘If an *exception* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.’’

```
w  =  1.0  /  0.0  ;      /*  infinity in IEEE-754  */
x  =  0.0  /  0.0  ;      /*  NaN in IEEE-754  */
y  =  +0.0  ;    /*  plus zero  */
z  =  -  y  ;    /*  minus zero:  Must this be –0.0?  May it be +0.0?  */
```

Are the above representable?

§ 4.5.1 (‘‘Treatment of Error Conditions’’) page 112, lines 11-12: ‘‘The behavior of each of these functions is defined for all representable values of its input arguments.’’

What about non-numbers? Are they representable? What is `sin(`*NaN*`);`? If you got a NaN as input, then you can return NaN as output. But, is it a domain error? Must `errno` be set to `EDOM`? The NaN already indicates an error, so setting `errno` adds no more information. Assuming NaN is not part of Standard C ‘‘representable’’, but the hardware supports it, then using NaNs is an extension of Standard C and setting `errno` need not be required, but is allowed. Correct?

§ 4.5.1 (‘‘Treatment of Error Conditions’’) on page 112, lines 20-27 says: ‘‘Similarly, a *range error* occurs if the result of the function cannot be represented as a `double` value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign (except for the `tan` function) as the correct value of the function; the value of the macro `ERANGE` is stored in `errno`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; whether the integer expression `errno` acquires the value of the macro `ERANGE` is implementation-defined.’’

What about denormal numbers? What is `sin(DBL_MIN / 3.0L);`? Must this be considered underflow and therefore return zero, and maybe set `errno` to `ERANGE`? Or may it return `DBL_MIN`/3.0, a denormal number? Assuming denormals are not part of Standard C ‘‘representable’’, but the hardware supports it, then using them is an extension of Standard C and setting `errno` need not be required, but is allowed. Correct?

What about infinity? What is `exp(`*INF*`);`? If you got an INF as input, then you can return INF as output. But, is it a range error? The output value is representable, so that says: no error. The output value is bigger than `DBL_MAX`, so that says: an error and set `errno` to `ERANGE`. Assuming infinity is not part of Standard C ‘‘representable’’, but the hardware supports it, then using INFs is an extension of Standard C and setting `errno` need not be required, but is allowed. Correct?

What about signed zeros? What is `sin(-0.0);`? Must this return –0.0? May it return –0.0? May it return +0.0? Signed zeros appear to be required in the model in § 2.2.4.2.2 on page 16.

What is `sqrt(-0.0);`? IEEE-754 and IEEE-854 (floating-point standards) say this must be –0. Is –0.0 negative? Is this a domain error?

§ 4.9.6.1 (‘‘The `fprintf` Function’’) on page 133, line 32 33 says: ‘‘(It will begin with a sign only when a negative value is converted if this flag is not specified.)’’

What is `fprintf(stdout, "%+.1f", -0.0);`? Must it be –0.0? May it be +0.0? Is –0.0 a negative value? The model on page 16 appears to require support for signed zeros.

What is `fprintf(stdout, "%f %f", 1.0/0.0, 0.0/0.0);`? May it be the IEEE-854 strings of ‘‘`inf`’’ or ‘‘`infinity`’’ for the infinity and ‘‘`NaN`’’ for the quiet NaN? Would ‘‘`NaNQ`’’ also be allowed for a quiet NaN? Would ‘‘`NaNS`’’ be allowed for a signaling NaN? Must the sign be printed? Signs are optional in IEEE-754 and IEEE-854. Or, must it be some decimal notation as specified by § 4.9.6.1 page 134, line 19? Does the locale matter?

§ 4.10.1.4 (‘‘The `strtod` Function’’) on page 152, line 2 3 says: ‘‘If the subject sequence begins with a minus sign, the value resulting from the conversion is negated.’’

What is `strtod("-0.0", &ptr);`? Must it be –0.0? May it be +0.0? The model on page 16 appears to require support for signed zeros. All floating-point hardware I know about support signed zeros at least at the load, store, and negate/complement instruction level.

§ 4.10.1.4 (‘‘The `strtod` Function’’) on page 152, lines 12-15 says: ‘‘If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`.’’

If `HUGE_VAL` is +infinity, then is `strtod("1e99999",&ptr);` outside the range of representable values, and a range error? Or is it the ‘‘nearest’’ of `DBL_MAX` and INF?

Interpretation:

Principles for C floating-point representation

(These principles are intended to clarify the use of some terms in the standard; they are not meant to impose additional constraints on conforming implementations.)

1. ‘‘Value’’ refers to the abstract (mathematical) meaning; ‘‘representation’’ refers to the implementation data pattern.
2. Some (not all) values have exact representations.
3. There may be multiple exact representations for the same value; all such representations shall compare equal.
4. Exact representations of different values shall compare unequal.
5. There shall be at least one exact representation for the value zero.
6. Implementations are allowed considerable latitude in the way they represent floating-point quantities; in particular, as noted in Footnote 10 on page 15, the implementation need not exactly conform to the model given in § 2.2.4.2.2 for ‘‘normalized floating-point numbers’’.
7. There may be minimum and/or maximum exactly-representable values; all values between and including such extrema are considered to ‘‘lie within the range of representable values’’.
8. Implementations may elect to represent ‘‘infinite’’ values, in which case all real numbers would lie within the range of representable values.
9. For a given value, the ‘‘nearest representable value’’ is that exactly-representable value within the range of representable values that is closest (mathematically, using the usual Euclidean norm) to the given value.

(Points 3 and 4 are meant to apply to representations of the same floating type, not meant for comparison between different types.)

This implies that a conforming implementation is allowed to accept a floating-point constant of any arbitrarily large or small value.

Request for Interpretation # 28 (X3J11 Doc. No. 91-009)

Question 1:

§ 3.3, page 39, lines 18-27 state some very important rules governing how a strictly conforming program can access the value of an object. The basic theme of the rules is that an object's value may only be accessed through an lvalue of the appropriate type. These rules are required to permit C programs to be optimized.

The rules depend on the ''declared type of the object''. This seems to make the rules not apply if the object was not declared, which is the case for an object allocated using `malloc()`.

Do the rules somehow apply to dynamically allocated objects? Is a compiler free to optimize the following function:

```
void  f(int  *x,  double  *y)
{
       *x  =  0;
       *y  =  3.14;
       *x  =  *x  +  2;
}
```

into the equivalent function:

```
void  f(int  *x,  double  *y)
{
       *x  =  0;
       *y  =  3.14;
       *x  =  2;/*  *x  known to be zero  */
}
```

Or must an optimizer prove that pointers are not pointing at dynamically allocated storage before performing such optimizations?

Interpretation:

Case 1: unions `f(&u.i, &u.d)`

§ 3.3.2.3, page 43, lines 5-11:

> ... if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined.

Therefore, an alias is not permitted and the optimization is allowed.

Case 2: declared objects `f((int *)&d, &d)`

§ 3.3, page 39, lines 18-27 list specific ways in which declared objects can be accessed. Therefore, an alias is not permitted and the optimization is allowed.

Case 3: any other, including `malloc`ed objects `f((int *)dp, dp)`

We must take recourse to intent. The intent is clear from the above two citations and from Footnote 36 on page 39:

> The intent of this list is to specify those circumstances in which an object may or may not be aliased.

Therefore, this alias is not permitted and the optimization is allowed.

In summary, yes, the rules do apply to dynamically allocated objects.

Request for Interpretation # 29 (X3J11 Doc. No. 91-016)

Question 1:

§ 3.1.2.6 says

> … two structure, union, or enumeration types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values.

I have one question and one clarification, both about compatibility between two `struct`/`union`/`enum` types declared in separate translation units.

(1) Was it the committee's intent that the two types must have the same tag (or both lack tags) to be compatible? As the standard is written, the following is legal:

One Translation Unit:

```
struct foo { int i; } x;
```

Another Translation Unit:

```
extern struct bar { int i; } x;
```

Recommendation: this seems like an accidental omission. To be compatible, the two types should have the same tag, or both lack tags. I would guess that such was the committee's intent.

(2) Clarification: the phrase ''two structure, union, or enumeration types'' should be written ''two structure types, two union types, or two enumeration types''. The current standard, interpreted literally, allows a structure and a union with identical member lists to be compatible, even though this is clearly not the intent of the committee.

One Translation Unit:

```
union foo { int i; } x;
union bar { int i, j; } y;
```

Another Translation Unit:

```
extern struct foo { int i; } x;
extern struct bar { int i, j; } y;
```

Interpretation:

§ 3.1.2.6 says (by omission) that tags do not have to be the same for structure, union, or enumeration types to be compatible in separate translation units. Tags are used in succeeding declarations to ensure that they are of the same type. They are not used for type compatibility.

Does ''two structure, union, and enumeration types'' mean ''two structure types, two union types, or two enumeration types''? Yes.

Request for Interpretation # 30 (X3J11 Doc. No. 91-017)

Question 1:

Reference: § 4.5.1 (‘‘Treatment of Error Conditions’’) page 112, lines 14-17:

> For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. … an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.

If `sin(DBL_MAX)` results in `errno` being set to `EDOM`, is this violation of the standard? If yes, what should be the result of this call?

Interpretation:

§ 4.5.1 does not give license for an implementation to set `errno` to `EDOM` for `sin(DBL_MAX)`. The mathematical function is defined for that argument value. While a conforming hosted implementation must not set `errno` to `EDOM` for this case, the standard imposes no constraint on the accuracy of the result value.

Request for Interpretation # 31 (X3J11 Doc. No. 91-018)

Question 1:

Referring to § 3.3, page 39, lines 15-17:

> If an *exception* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

and § 3.4, page 56, lines 11-12:

> Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

What should be the result of the constant expression:

```
INT_MAX  +  2
```

— is this a constraint violation, or it should be mapped onto the set of representable values?

What should be the result of:

```
INT_MAX  +  2ul
```

How should compilers that do not evaluate the constant expressions at compile time behave?

What is the result of:

```
(INT_MAX*4)/4
```

Referring to § 3.5.2.2, page 62, lines 29-30:

> The expression that defines the value of an enumeration constant shall be an integral constant expression that has a value representable as an `int`.

What is the result of:

```
enum  {  a=INT_MAX,  b  };
```

Does this violate the C standard?

Interpretation:

`case INT_MAX + 2:` is a constraint violation.

`case INT_MAX + 2ul:` is okay, representable.

`case (INT_MAX*4)/4:` is a constraint violation.

When § 3.4 says on page 56, lines 11-12

> Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

the committee's judgement of the intent is that the ''representable'' requirement applies to each subexpression of a constant expression, as shown in the third example. A constant expression is meant as defined by the syntax rules.

`enum { a=INT_MAX, b };` is a constraint violation.

Request for Interpretation # 33 (X3J11 Doc. No. 91-037)

Question 1:

Is a conforming implementation required to diagnose all violations of ‘‘shall’’ and ‘‘shall not’’ statements in the standard, even if those statements occur outside of a section labeled ‘‘Constraints’’?

An example that illustrates this question is:

```
struct s { char field:1; };
```

This fragment violates a statement in § 3.5.2.1 on page 61, line 30: ‘‘A bit-field shall have a type that is a qualified or unqualified version of one of `int`, `unsigned int`, or `signed int`.’’ Must a conforming implementation issue a diagnostic for this violation of ‘‘shall’’?

Following are two different ways in which the standard has been interpreted. These interpretations came up during discussions over NIST conformance tests for an ANSI-C FIPS. I would like to ask X3J11 for an interpretation of this issue, perhaps based on one or both of the interpretations given.

Suggested interpretation # 1 —

> § 1.6 (‘‘Definitions of Terms’’) states in the very beginning: ‘‘In this standard, ‘shall’ is to be interpreted as a requirement on an implementation or on a program; conversely, ‘shall not’ is to be interpreted as a prohibition.’’
>
> Therefore *every* ‘‘shall’’ is viewed as testable. The question is what happens if a ‘‘shall’’ is violated.
>
> § 2.1.1.3 (‘‘Diagnostics’’) provides the answer: ‘‘A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of *any syntax rule* or *constraint*. Diagnostic messages *need not be* produced in other circumstances.’’ [emphasis added]
>
> Therefore every violation of a ‘‘shall’’ should be treated as a failure to meet the requirements of the standard (first definition). Any violation of syntax rules, semantic rules, or sections labeled as ‘‘Constraints’’ should therefore generate a diagnostic.
>
> According to this interpretation, a diagnostic should be produced for the example given above.

Suggested interpretation # 2 —

> § 2.1.1.3 states that diagnostics must be produced ‘‘for every translation unit that contains a violation of any syntax rule or constraint. Diagnostic messages need not be produced in other circumstances.’’
>
> *Syntax rules* are those items listed in the ‘‘Syntax’’ sections of the standard. *Constraints* are those items listed in the ‘‘Constraints’’ sections of the standard.
>
> The standard specifies in § 1.6, page 3, lines 42-43 that when the words ‘‘shall’’ or ‘‘shall not’’ appearing outside of a constraint are violated, the behavior is undefined.
>
> For undefined behavior, the standard specifies in § 1.6, page 3, lines 36-37 that ‘‘the standard imposes no requirements’’. Thus a conformance suite should not test for the words ‘‘shall’’ or ‘‘shall not’’ outside of a Constraints section, since the standard imposes no requirements.
>
> According to this interpretation, the standard imposes no requirements on a conforming implementation for the program fragment above. A conforming implementation could choose to accept this program (see also Footnote 6 to § 2.1.1.3 on page 7), it could issue a diagnostic, or have any other behavior.

Interpretation:

Concerning a violation of § 3.5.2.1 Semantics , page 61, line 30: No diagnostic is required; this is undefined behavior. It is not a violation of a constraint or syntax.

Concerning a violation of § 1.6, page 2, lines 32-33: No diagnostic is required.

Suggested interpretation # 2 is the correct one.

This is X3J11's interpretation of the ANSI standard. Conformance to FIPS is beyond the scope of X3J11. We can't comment on this nor endorse a sentence that begins ''Thus a conformance suite should not test …''

Request for Interpretation # 34 (X3J11 Doc. No. 91-038)

Question 1:

In the ‘‘C Users Journal’’, Vol. 8 No. 7, July 1990, P. J. Plauger gives the following example on page 10:

```
extern  int  a[];
int  f()  {
       extern  int  a[10];
       ...
       }
int  sizea  =  sizeof  a;        /*  error  */
```

Mr. Plauger claims that the size information from the inner scope ‘‘evaporates’’ when its scope ends, and the operand to the `sizeof` operator has an incomplete type.

We cannot find unequivocal support for this claim in the standard.

§ 3.1.2.2 says on page 22, lines 10-11

> … each instance of a particular identifier with *external linkage* denotes the same object or function.

Combining § 3.1.2.6 and § 3.5.4.2, we find that the two declarations for `a` are compatible and we may construct a composite type. The composite type is ‘‘array of 10 `int`s’’.

§ 3.1.2.6 on page 26 lines 19-20 discusses the case of two declarations in the same scope, but does not discuss the case of two declarations for the same object in different scopes.

But § 3.1.2.5 says on page 25, lines 8-9:

> An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage).

The identifier `a` appears in two declarations, and denotes the same object. The second declaration completes the type for the identifier in the inner scope. The two identifiers denote the same object, so it would seem reasonable to say the type of that object is completed.

Is the size information in the inner scope lost upon leaving the scope?

Interpretation:

Yes, the size information is lost upon leaving the scope. As pointed out in the foregoing, the composite type is only for the same scope. § 3.1.2.5 page 25, lines 8-9 have an implied ‘‘in the same scope’’.

Question 2:

If no size information is known in the outer scope, then consider the following example:

```
extern  int  a[];
int  f()  {
       extern  int  a[10];
       ...
       }
int  g()  {
       extern  int  a[20];       /*  error?  */
       ...
       }
```

Is this legal? If not, does it violate a constraint?

Interpretation:

No, the example exhibits undefined behavior. It does not violate a constraint. § 3.1.2.2 page 22, lines 10-13 describe ‘‘same object’’; § 3.1.2.6 page 26 lines 9-10 require that ‘‘All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.’’

Request for Interpretation # 35 (X3J11 Doc. No. 91-039)

Question 1:

```
void  f(a,  b)
int  a(enum  b  {x,  y};);
int  b;
{
}
```

Now this example is perverse because a prototype declaration is used to declare the parameter of an old-style function declaration. But anyway…

Is the declaration of the parameter `a` legal or a constraint error?

Now `a(...)` is a declarator.

§ 3.7.1 says on page 83, lines 7-8:

> … each declaration in the declaration list shall have at least one declarator, and those declarators shall declare only identifiers from the identifier list.

The identifier list contains `a` and `b`.

The declarator for parameter `a` declares the identifiers `a`, `b`, `x`, and `y`.

`b` is in the identifier list, so that is okay. But `x` and `y` are not. Constraint error (methinks so)?

See § 3.1.2, page 20 for a definition of an identifier.

Interpretation:

There is no constraint violation. The scopes of `b`, `x`, and `y` end at the right-parenthesis at the end of the `enum`, so there is no violation. It is difficult to *call* the function `f`, but there is no constraint violation. The phrase ‘‘each declarator declares one identifier’’ in § 3.5.4 refers to `a`, not to `b`, `x`, or `y`.

Question 2:

Also consider:

```
void  g(c)
enum  m{q,  r}  c;
{
}
```

What is the scope of `m`, `q`, and `r`?

§ 3.1.2.1 says on page 21, lines 28-29: ‘‘… appears outside of any block or list of parameters, the identifier has *file scope*, …’’

It says on page 21, lines 30-31: ‘‘… appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, …’’

Now the above three identifiers appear outside of any block or list of parameters but they are within the list of parameter declarations.

Who wins?

Interpretation:

The scope of `m`, `q`, and `r` ends at the close-brace (block scope). The operative wording is the more specific statement on page 21, lines 30-31: ‘‘… appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, …’’